UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Simulación de fluidos en aplicaciones de tiempo real
Autores: Ramiro Fages de la Canal
Director: Oscar N. Bría
Codirector: -
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

Esta tesina aborda la simulación de fluidos en tiempo real utilizando la GPU, enfocándose en el modelado del humo. Comienza presentando los conceptos teóricos relacionados con la misma, partiendo de las ecuaciones de Navier-Stokes y finaliza con la visualización en pantalla de los resultados. La implementación de la simulación y visualización se realiza en dos dimensiones utilizando la GPU, junto con las APIs de DirectCompute y Direct3D.

Palabras Claves

Simulación, fluidos, tiempo real, GPU, Navier-Stokes, backwards-advection, Direct3D, DirectCompute, CUDA.

Trabajos Realizados

Se realizó una investigación acerca de los fundamentos físicos y matemáticos involucrados en el comportamiento y simulación de fluidos. Se introdujo la arquitectura de la GPU sobre la cual se implementa la etapa de la simulación, así como también consideraciones de performance que deben tenerse en cuenta. Luego se presentó el funcionamiento del rendering pipeline necesario para poder realizar la visualización de los resultados de la simulación. Por último se implementó una pequeña aplicación integrando todos los conceptos previamente mencionados.

Conclusiones

Se ha demostrado que la simulación de fluidos en tiempo real es un área que aún está en constante crecimiento, y que tiene un gran potencial para su uso en la industria cinematográfica, científica y de videojuegos.

Se presentaron los conceptos físicos necesarios para poder entender las ecuaciones de los fluidos, así como también la arquitectura de la GPU y el rendering pipeline para poder realizar una implementación.

Trabajos Futuros

Aumentar el grado de calidad general del fluido utilizando un mejor método de advección como por ejemplo MacCormack. Reducir el tiempo necesario para la convergencia en el cómputo de la presión, utilizando el método de multi-grillas. Aumentar el detalle del humo, utilizando el método de Vorticity Confinement.
Llevar la simulación a tres dimensiones, implementando un algoritmo de raymarching para la visualización del volumen.

Fecha de la presentación: OCTUBRE 2017

CONTENIDOS

Contenidos II
Lista de figuras III

1. **Resumen**.....**1**
2. **Introducción**.....**2**
3. **Contexto**.....**4**
   - 3.1. Industria cinematográfica.....4
   - 3.2. Industria de los videojuegos.....7
   - 3.3. Ambito cientifico.....10
4. **Motivación**.....**1**
   1
5. **Objetivo**.....**13**
6. **Estado del arte**.....**14**
7. **Simulación de fluidos: Teoría**.....**16**
   - 7.1. Introducción.....16
   - 7.2. Resolviendo las ecuaciones de Navier-Stokes.....28
     - 7.2.1. Advección.....36
     - 7.2.2. Adición de fuerzas.....38
     - 7.2.3. Proyección de la presión.....38
     - 7.2.4. Condiciones iniciales y de frontera.....41
8. **Simulación de fluidos: Implementación**.....**43**
   - 8.1. Simulación.....43
     - 8.1.1. GPU.....43
     - 8.1.2. DirectCompute.....55
     - 8.1.3. Implementación de la simulación.....58
     - 8.1.4. Consideraciones de performance.....65
   - 8.2. Visualización.....67
     - 8.2.1. Introducción al rendering pipeline.....67
     - 8.2.2. Etapas del rendering pipeline.....71
     - 8.2.3. Implementación de la visualización.....83
9. **Conclusiones**.....**89**
10. **Trabajos futuros**.....**90**

11. Bibliografía **91**
APÉNDICE A - Glosario **95**
APÉNDICE B - Operadores del cálculo vectorial **96**

LISTA DE FIGURAS

1. Imagen tomada de la herramienta Blender.....5

2 Relación entre la textura y el modelo.....5

3 Comparación entre una escena renderizada con rasterización, y con ray tracing.....6

4 Técnica de renderizado utilizando ray tracing.....6

5 Imagen tomada de la herramienta Unity3D, visualizando el modelo de un cubo importado en la escena (nótese la similitud con Blender).....8

6 Rasterización de un triángulo (en negro), con los fragmentos a ser generados(en verde), y el centro de cada fragmento (en celeste).....9

7 Ejemplo de un velocity field.....18

8 Flujo entrante y saliente de una celda.....18

9 Flujo entrante y saliente para distintos puntos del fluido.....19

10 In-flux y out-flux.....19

11 Flujo sin divergencia.....20

12 Flujo con divergencia.....21

13 A la izquierda cell-centered grid, y a la derecha staggered grid.....27

14 Cálculo de la divergencia.....29

15 Método de proyección.....33

16 Actualización de la velocidad u de una celda (indicada en rojo).....35

17 Corrección de la velocidad u aplicando la condición de Dirichlet.....39

18 Este tema será discutido nuevamente durante la implementación de las ecuaciones.....40

19 Grid de 3x2 bloques. Cada bloque con 3x2 threads. 36 Threads en total.....43

20 Situación A (izquierda) y B(derecha).....46

21 Situación C (izquierda) y D (derecha).....47

22 Situación E, en donde se produce un conflicto en los bancos 1 y 3.....47

23 Memoria global.....48

24 Acceso no alineado a dos líneas de caché....49

25 Mapeo de threads a texeles de una imagen....50

26 Acceso al elemento (1,1) de la textura utilizando coordenadas en el rango 0..1....51

27 Muestreo de textura configurada para repetir coordenadas fuera de rango....52

28 Muestreo de textura configurada para restringir las coordenadas fuera de rango....52

29 Relación entre DirectCompute y la arquitectura de CUDA....53

30 Etapas y sub etapas que componen la simulación....57

31 Consumo de memoria para diferentes resoluciones en una textura 3D de 4 canales (RGBA) en punto flotante con 16 bits de precisión....64

32 Etapas del rendering pipeline (simplificado)....67

33 Common shader core....68

34 Subdivisión de un cuadrado en dos triángulos....70

35 Etapas del rendering pipeline....70

36 Dos líneas compuestas por dos vértices cada una....71

37 Tres líneas compuestas por cuatro vértices....72

38 Tres líneas compuestas por seis vértices....72

39 Amplificación de datos durante la rasterización....74

40 Transformación de espacios de coordenadas....75

41 Primitive culling....76

42 Relación entre el vector normal del triángulo y el orden de los vértices....76

43 Backface culling....76

44 Backface culling....77

45 Proceso de primitive clipping....77

46 Interpolación de los colores de los vértices v1 y v2, a lo largo de los fragmentos f0..f5....78

47 Geometría resultante, compuesta por dos triángulos....82

48 Mapeo de coordenadas UV de los vértices a una imagen....85

1. RESUMEN

Esta tesina aborda la simulación de fluidos en tiempo real utilizando la GPU, enfocándose en el modelado del humo. Comienza presentando los conceptos teóricos relacionados con la misma, partiendo de las ecuaciones de Navier-Stokes y finaliza con la visualización en pantalla de los resultados. La implementación de la simulación y visualización en la GPU se hace en dos dimensiones, haciendo uso de la API de DirectCompute y Direct3D.

La implementación utiliza el método de backwards-advection presentado por Jos Stam para realizar la advección del fluido. Para el cómputo de la presión se utiliza el método iterativo de Jacobi. El procesamiento y almacenamiento de los valores utilizados durante la simulación es llevado a cabo mediante el uso de texturas, las cuales son compartidas entre DirectCompute y Direct3D para facilitar su visualización.

2. INTRODUCCIÓN

La simulación en tiempo real de fenómenos naturales es hoy un área de gran interés, debido principalmente a las demandas de industria cinematográfica y de los videojuegos, aunque también se pueden encontrar sus usos en aplicaciones científicas.

Dentro de los fenómenos naturales es posible identificar a aquellos cuyo comportamiento puede ser modelado como el de un fluido. Por ejemplo:

- Agua fluyendo en un río
- Fuego saliendo de una antorcha
- Vapor
- Humo de cigarrillo
- Lava
- Nubes

La correcta reproducción de dichos fenómenos agrega un gran valor a la percepción de los objetos y enriquece los paisajes de la escena que se quiera visualizar en una película o en un videojuego. Es por esto que particularmente la industria cinematográfica tuvo un gran interés en la simulación de fluidos desde que Jos Stam presentó su publicación titulada Stable Fluids en 1999**[1],** en la cual se detallaba un método que permitía realizar una simulación incondicionalmente estable[1]. Sin embargo debido a las limitaciones de hardware en aquellas épocas, no cualquier tipo de aplicación podría realizar una implementación de la simulación que corra en tiempo real.

Normalmente cuando se habla de una simulación de fluidos, también hay asociada una etapa que involucra la visualización de los resultados de dicha simulación. Dependiendo del tipo de aplicación que se deba desarrollar, tanto la simulación como la visualización pueden ser ejecutadas en tiempo real u off-line. Durante el desarrollo de esta tesina cuando se mencione únicamente a la simulación, también se deberá tener en cuenta la etapa de visualización asociada a excepción de los casos en donde se especifique lo contrario.

El algoritmo propuesto por Stam no era apto para ser utilizado en los videojuegos dado que el algoritmo para la simulación (en ese entonces) debía ser implementada en la CPU, y los videojuegos contaban[2] con grandes restricciones con respecto al tiempo que se disponía para actualizar cada uno de sus subsistemas (inteligencia artificial, simulación de físicas, renderización, etc). Si el videojuego se propone actualizar su estado a una velocidad de 60 fotogramas (frames) por segundo, quiere decir que se disponen de 16.66 milisegundos para realizar todas las tareas necesarias, incluyendo la simulación y renderización (visualización) del fluido.

En el ámbito de la industria cinematográfica, las películas no necesitan implementar la simulación o renderizado en tiempo real[3]. El flujo de trabajo típico en la producción de una

[1] En este contexto se dice incondicionalmente estable al ser independiente del time-step utilizado para avanzar en la simulación.
[2] Estas restricciones de tiempo hoy en día siguen siendo las mismas.
[3] Pero pueden obtener grandes ventajas si lo hacen.

película consiste en tomar cada una de las imágenes capturadas por una cámara o generada por computadora, y procesarlas utilizando distintas técnicas de iluminación o efectos especiales. El procesamiento que se realiza sobre cada una de las imágenes puede tardar desde segundos hasta horas en finalizar, para luego componer todas las imágenes resultantes en lo que será el video final. Las simulaciones de fluidos son ampliamente utilizadas en la industria cinematográfica y de efectos especiales, principalmente debido a la ventaja de no contar con fuertes restricciones de tiempo.

Las aplicaciones científicas pueden tener o no la necesidad de realizar la simulación en tiempo real, pero requieren que dicha simulación sea precisa en cuanto al comportamiento de los fluidos. Es decir, si se realiza una simulación para analizar el perfil aerodinámico de un avión en un túnel de viento virtual, es de vital importancia que los resultados sean precisos sin importar la calidad visual (siempre y cuando los detalles puedan ser observados). Al contrario, en las películas se busca particularmente una visualización realista con un cierto grado de precisión en su comportamiento.

Es importante destacar que el hecho de que un algoritmo corra en la CPU no significa que no pueda correr en tiempo real. A lo largo de este trabajo cuando se habla de real time rendering (renderizado en tiempo real), se referirá al renderizado a una velocidad de 25 o más frames por segundo (fps). Los juegos deben correr en tiempo real para proporcionar una experiencia fluida al usuario y por lo tanto se intenta mantener el framerate (tasa de frames por segundo) por encima de los 25 fps.

Cuando la velocidad varía entre 1 y 24 fps se lo considerará como interactive rendering (renderizado interactivo), y suele ser utilizado en aplicaciones donde es importante tener algún tipo de respuesta sin importar demasiado su velocidad o el retardo que hay entre las acciones del usuario y la respuesta del sistema.

El renderizado a una velocidad menor a 1 fps se lo considerará como offline rendering, comúnmente utilizado por las películas y aplicaciones científicas en donde no se requiere de una respuesta durante el proceso de renderizado.

Por lo tanto la simulación de fluidos puede llegar a ejecutarse en tiempo real sobre la CPU, pero utilizando una baja calidad[4], la cual hace que el resultado no pueda apreciarse correctamente y por lo tanto no tenga un uso práctico.

Con la llegada y el rápido crecimiento de las GPUs, se comenzó a considerar su uso en los algoritmos para la simulación de fluidos **[2]**, los cuales gracias a su propiedad de ser altamente paralelizables fueron capaces de conseguir una notable reducción en el tiempo que toma ejecutarse la simulación en altas resoluciones, permitiendo así su uso en aplicaciones de tiempo real con un alto grado de calidad.

[4] En los siguientes capítulos se hará mención de a qué se refiere con baja calidad.

3. CONTEXTO

Antes de continuar con los siguientes capítulos es necesario analizar el contexto sobre el cual se desarrollarán los contenidos de esta tesina, para poder así comprender el lugar que ocupa cada una de las herramientas y conceptos presentados.

A continuación se hará una breve descripción del flujo de trabajo típico utilizado en las herramientas tanto de la industria cinematográfica como de los videojuegos, en otras, la serie de pasos realizados por un artista (o diseñador de efectos especiales) en cada herramienta que conlleva a la creación de una imagen.

Luego se describirá brevemente los usos, ventajas y problemas que existen en las aplicaciones científicas que desean utilizar la simulación de fluidos en tiempo real.

3.1 INDUSTRIA CINEMATOGRÁFICA

En el ámbito de las películas, se pueden distinguir a aquellas que utilizan CGI (Imágenes generadas por computadora) para diversos propósitos. Una película con actores reales puede utilizar CGI para la creación de VFX (efectos visuales), como por ejemplo explosiones, los cuales serán agregados en la etapa de post-producción. Por otro lado también se encuentran aquellas películas creadas en su totalidad utilizando CGI. Algunos ejemplos son Shrek (Dreamworks) y Toy Story (Pixar).

Por cuestiones de simplicidad, de ahora en adelante se considerarán únicamente a las películas creadas enteramente con CGI, sin embargo el flujo de trabajo mencionado a continuación es similar al de aquellas películas que utilizan CGI para la creación de VFX.

Para la creación de dichas películas, existen diversas herramientas (Autodesk Maya, Blender, Cinema4D) que permiten, mediante una interfaz gráfica, la creación de los objetos/modelos y efectos especiales que compondrán la escena, así como también el renderizado de la misma.

El usuario podrá elegir entre crear o importar (mediante un formato que comparta con otras herramientas similares, por ejemplo .obj o .fbx) los modelos que compondrán la escena. Estos modelos están compuestos por una serie de polígonos, los cuales a su vez están conformados por un conjunto de vértices interconectados.

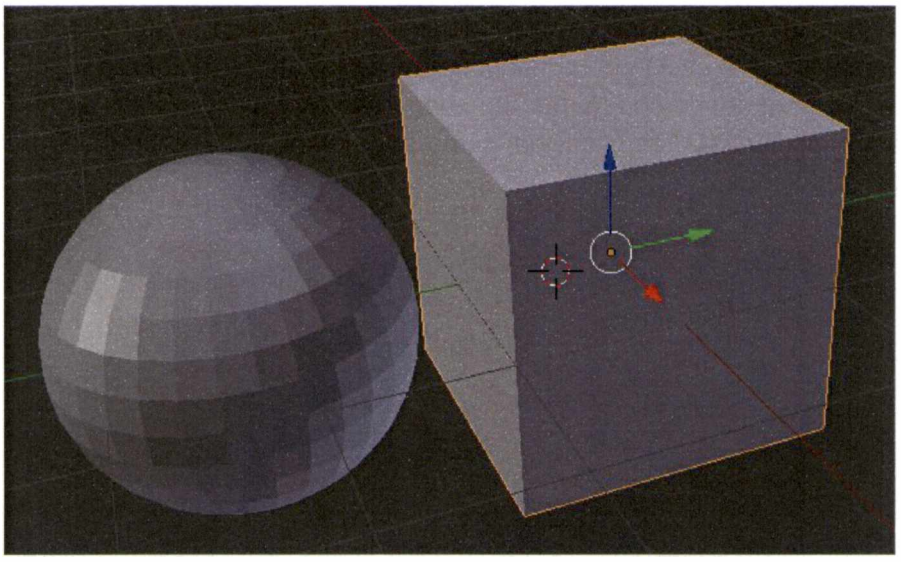


Figura 1: Imagen tomada de la herramienta *Blender.*

Con los objetos en la escena, es posible crear (o importar) animaciones para los mismos siempre que la herramienta lo permita, así como también asignar a los modelos lo que se conoce como textura (una imagen que contendrá la apariencia del modelo a la que se la aplicará). Por ejemplo si el modelo representa la cabeza de un personaje, en la imagen se pintarán los ojos, boca, y demás rasgos faciales, así como también el color de piel. Esta imagen es normalmente creada con una herramienta externa, como por ejemplo Adobe Photoshop.

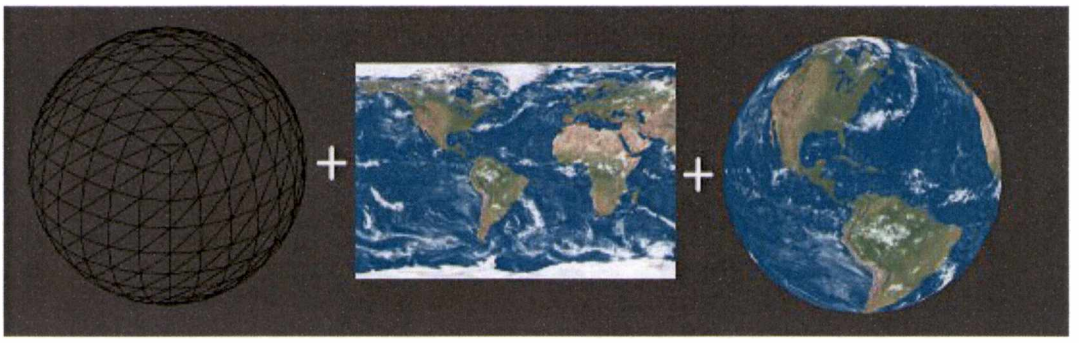


Figura 2: Relación entre la textura y el modelo.

Por último, teniendo los objetos y sus respectivas animaciones en su lugar, el artista procederá a configurar la iluminación de la escena y demás efectos especiales, para luego realizar el renderizado final de uno o varios frames dependiendo de si desea visualizar la secuencia de una animación, o simplemente ver la apariencia final de la escena.

Hasta este momento, el artista visualiza la escena de forma simplificada desde el punto de vista de la calidad visual. Por ejemplo si bien se puede modificar la iluminación de la escena, este cambio no se verá reflejado de forma inmediata[5]. Esto se debe a que mientras

[5] Esto dependerá del modelo de iluminación que se use, pero para este ejemplo se asume que se utiliza un modelo de iluminación complejo.

se trabaja en el editor[6] la escena es renderizada mediante la rasterización utilizando la GPU, mientras que la creación de la imagen final se la renderiza utilizando técnicas de ray tracing y será allí en donde se tomarán en cuenta varios aspectos visuales que no pueden representados mediante la rasterización, como por ejemplo reflejos precisos en los objetos, o iluminación ambiental.

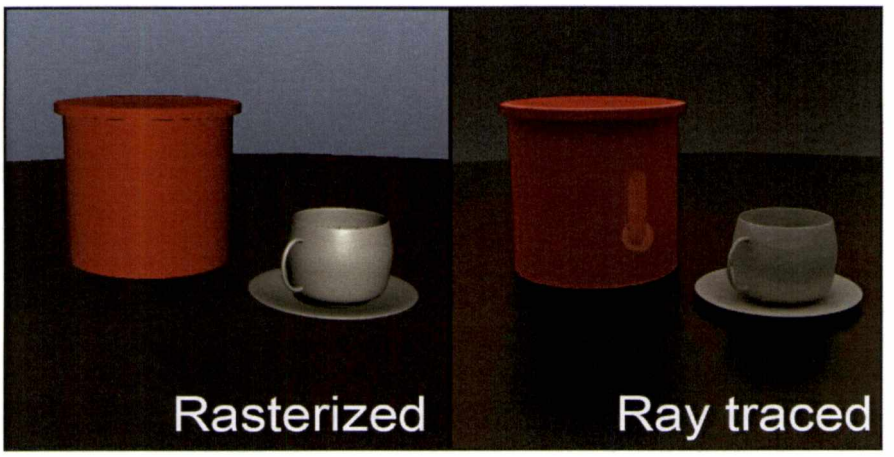


Figura 3: Comparación entre una escena renderizada con rasterización, y con ray tracing.

El renderizado utilizando ray tracing consiste en "disparar" un rayo desde un punto de origen (la cámara) a través de cada pixel en la imagen que se está componiendo, luego se realizará una prueba de intersección entre cada uno de los rayos disparados y todos los objetos de la escena. Si el rayo intersecta con uno o más objetos, se retornará el color del objeto mas cercano con el que está intersectando, de lo contrario se retornará negro (por defecto).

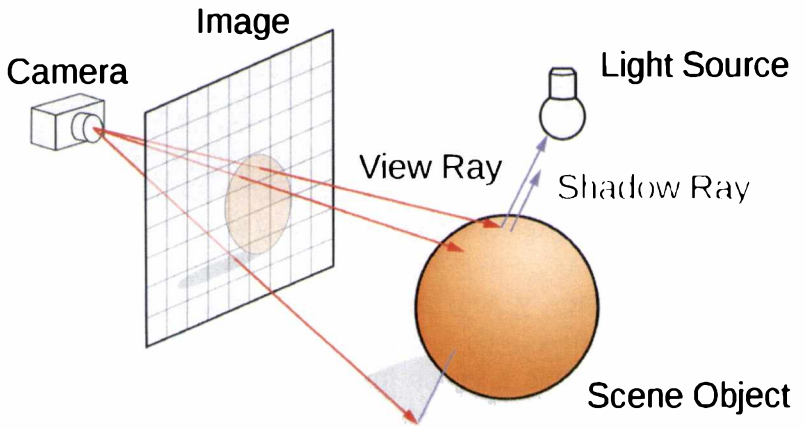


Figura 4: Técnica de renderizado utilizando ray tracing.

[6] El editor es la aplicación que le proveerá todas las herramientas al usuario para poder importar objetos, ubicarlos en la escena, modificarlos y realizar el renderizado de un frame (entre otras cosas). En otras palabras, el usuario trabaja el 90% del tiempo en el editor.

Esto quiere decir que si se intenta renderizar una imagen con una resolución de 256x256 píxeles, se dispararán 256x256 rayos, y cada uno de los rayos realizará un chequeo de colisión con cada uno de los objetos en la escena (si el objeto es un modelo compuesto de triángulos, se realizará el chequeo con cada uno de los triángulos), de forma tal que el tiempo de ejecución del algoritmo crece de forma exponencial con respecto a la resolución y a la cantidad de objetos en la escena, por lo tanto esta es una técnica costosa.

Típicamente las implementaciones suelen ser sobre una CPU, la cual no es capaz de generar imágenes en tiempo real para escenas de complejidad típica, y por lo tanto caen en la categoría de offline rendering, sin embargo actualmente se están haciendo avances en el área de real time ray tracing utilizando la GPU para su implementación (Nvidia Optix **[3]**).

Esta es una técnica muy simple de implementar en la CPU y proporciona gran flexibilidad al momento de generar imágenes fotorealistas, ya que permiten modelar el comportamiento de la luz interactuando con distintos objetos y materiales. Por ejemplo, si se desea agregar materiales reflectivos, todo lo que se debe hacer al encontrar una intersección, es disparar un segundo rayo partiendo del punto de intersección y en la dirección del reflejo con respecto a la superficie, retornando el color de aquel objeto con el que intersectó (si es que existe) con este segundo rayo.

Las técnicas de renderizado utilizando ray tracing proporcionan resultados precisos con alto nivel de realismo, con un alto costo en su tiempo de ejecución. Es por esto que normalmente el renderizado de las imágenes deja al artista segundos, minutos o incluso horas esperando hasta ver el resultado final.

En conclusión, el artista pasará la mayor parte de su tiempo trabajando con los objetos en la escena, utilizando el editor que le provee la herramienta para realizar las modificaciones necesarias. Una vez que está conforme con la composición de la escena, pasará a realizar los ajustes visuales necesarios, valiéndose del renderizado por ray tracing para observar el resultado final, modificando valores y repitiendo el proceso hasta alcanzar los resultados esperados.

3.2 INDUSTRIA DE LOS VIDEOJUEGOS

La creación de videojuegos conlleva requerimientos distintos a los de la creación de una película, sin embargo puede verse que aun así comparten varias similitudes.

Las herramientas disponibles hoy en día para la creación de videojuegos, conocidas como game engine, proveen un editor con interfaz gráfica que permite importar distintos tipos de recursos, como por ejemplo imágenes, modelos, animaciones, sonidos, y demás, de la misma forma que en la producción de una película. Ejemplos de game engines de hoy en día son Unity3D, Unreal Engine, Torque 3D, Cocos2D, CryEngine.

Luego un programador mediante el lenguaje de programación que provea el game engine, utilizará los recursos importados para construir las escenas y jugabilidad con la que contará el juego.

A diferencia de las herramientas de la industria cinematográfica, el game engine no provee la capacidad de creación de recursos[7], sin embargo otorga un gran grado de flexibilidad para posicionar objetos en la escena, y realizar así la composición de la misma forma que en la creación de una película.

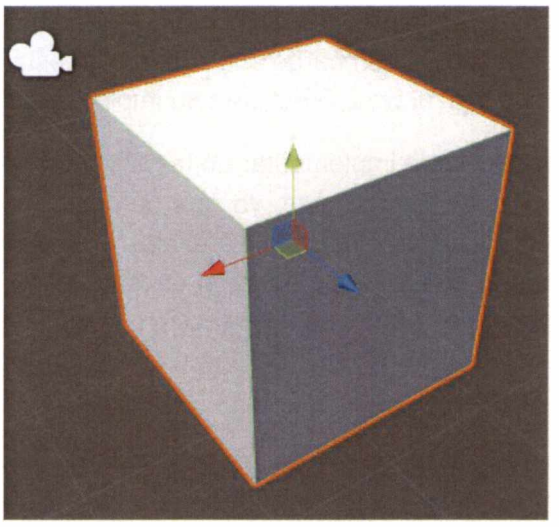


Figura 5: Imagen tomada de la herramienta Unity3D, visualizando el modelo de un cubo importado en la escena (nótese la similitud con Blender).

El propósito de las herramientas de la industria cinematográfica es el de generar imágenes, que luego serán utilizadas para producir un video. El propósito de los game engines es generar una aplicación (el videojuego o aplicaciones interactivas) la cual pasará a correr en la computadora de los usuarios finales, y por lo tanto el desempeño de la aplicación es de vital importancia para asegurar una buena experiencia a los usuarios.

Dado que el juego deberá correr en tiempo real, no es posible utilizar la renderización mediante ray tracing, por lo tanto se deberá utilizar la rasterización como principal método de visualización. Esto tiene la ventaja de que la visualización de la escena dentro del editor es la misma que se verá durante la ejecución del juego, lo cual facilita la configuración e iteración de los aspectos visuales.

A continuación se describirá de forma simplificada el funcionamiento de la rasterización, con el objetivo de entender a qué se debe su amplio uso en las aplicaciones que requieren de la visualización de objetos 3D (o incluso 2D) en tiempo real.

La rasterización, hecha por una unidad de hardware en la placa de video, consiste en la transformación de polígonos (puntos, líneas, y triángulos únicamente) en fragmentos. Un fragmento es un potencial pixel que se mostrará en pantalla si es que sobrevive a las etapas

[7]Aunque si se lo desea, se podría crear una extensión de la herramienta que lo permita.

restantes del renderizado gráfico, las cuales podrían descartar el fragmento de ser necesario.

Por ejemplo si se desean rasterizar dos puntos con exactamente la misma posición en el espacio, pero un punto estando "mas cerca" de la cámara[8], el rasterizer generará dos fragmentos, y descartará aquel que se corresponda con el punto más lejano a la cámara (ya que estará siendo tapado por uno más cercano a la misma).

Todos aquellos fragmentos que sobrevivan a las etapas de descarte, pasarán a convertirse en pixeles al momento de ser mostrados en la pantalla.

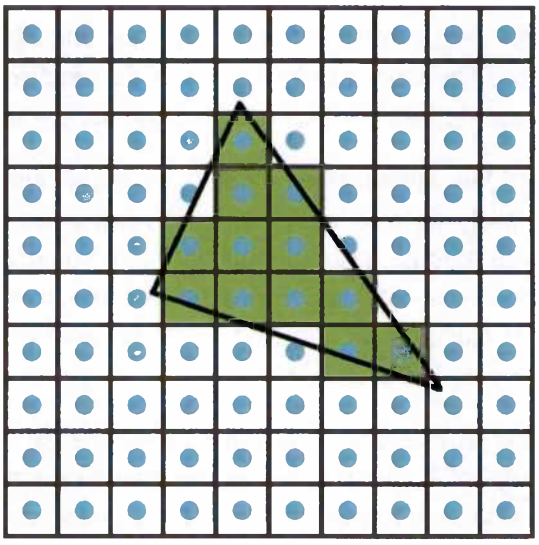


Figura 6: Rasterización de un triángulo (en negro), con los fragmentos a ser generados(en verde), y el centro de cada fragmento (en celeste).

Como se puede observar en la imagen de arriba, para el caso de los triángulos se generarán sólo aquellos fragmentos en donde el polígono cubra su centro[9].

A diferencia del raytracing, la rasterización no tiene noción de los conceptos de "escena" o "modelos". Todo lo que hace es recibir una serie de polígonos y generar los fragmentos necesarios, lo cual impone grandes limitaciones en los algoritmos que se pueden utilizar para generar imágenes realistas (utilizando sombras, reflejos, etc). Sin embargo gracias a esta simplicidad es posible diseñar una unidad de hardware en la GPU para encargarse específicamente de la rasterización, permitiendo que se pueda visualizar una gran cantidad de objetos en tiempo real.

En conclusión, gracias a la rasterización por hardware es posible renderizar escenas con una gran cantidad de objetos 3D en tiempo real, siendo la técnica utilizada tanto en el editor de un game engine como en la aplicación durante su ejecución.

[8] Por defecto, un punto ocupa un pixel en pantalla.
[9] Este comportamiento puede cambiar en el caso de utilizar multi-sampling anti aliasing.

3.3 AMBITO CIENTIFICO

Las simulaciones de fluidos en las aplicaciones de tiempo real en el ámbito científico, como por ejemplo simulaciones aerodinámicas de túneles de viento virtuales, cuentan con la necesidad de una mayor precisión en los resultados sin darle demasiada importancia a la representación visual que se utilice. Además, no sólo se requiere de mayor precisión sino que también aumenta la complejidad de los objetos de interés sobre los que se desea realizar la simulación, como por ejemplo simular el flujo de viento sobre la superficie de un auto **[39]**.

Este aumento en el nivel de complejidad hace que se dificulte el desarrollo de dichas aplicaciones, muchas veces llevando a reducir el nivel de precisión, o a utilizar geometrías en dos dimensiones en lugar de tres para reducir la magnitud de los datos **[4]**.

Habiendo dicho esto, en las aplicaciones científicas sigue siendo deseable el uso de simulaciones en tiempo real, ya que se provee la ventaja de poder interactuar con la simulación mientras ésta se ejecuta, permitiendo por ejemplo agregar o mover obstáculos y visualizar de forma inmediata la evolución del fluido en respuesta a estos cambios. Otra ventaja es la de poder realizar una exploración en el espacio de parámetros, permitiendo realizar rápidamente ajustes en la simulación.

Las implementaciones que se pueden encontrar hoy en día **[5]** utilizan las tecnologías de GPGPU como Cuda u OpenCL, junto con las de visualización como OpenGL. Para la implementación, se ha vuelto muy popular el uso de las ecuaciones (y métodos numéricos) de Lattice Boltzmann **[6]**. Estas ecuaciones satisfacen las ecuaciones de Navier-Stokes para flujos incompresibles, y tienen la particularidad de haber sido diseñadas especialmente para su ejecución en arquitecturas paralelas. Además, son computacionalmente más demandantes, haciendo que se aprovechen mejor los recursos en términos de poder de procesamiento[10]. Estas ventajas hacen que los métodos de Lattice Boltzmann sean adecuados para una implementación en la GPU.

[10] La implementación de las ecuaciones de Navier-Stokes son computacionalmente simples, pero requieren una alta transferencia de memoria, desaprovechando el poder de cómputo que se ofrece en una GPU.

4. MOTIVACIÓN

Como se mencionó en la introducción, la representación de fenómenos naturales mediante la simulación de fluidos provee el realismo necesario que las industrias de los videojuegos y cinematográfica desean obtener.

Las herramientas utilizadas en la producción de películas e imágenes generadas por computadora (CGI) suelen ser tediosas al momento de trabajar con simulación de fluidos, debido en parte a que los algoritmos que componen la simulación son de alta complejidad y a que son implementados en la CPU, lo cual sumado al hecho de que el renderizado de la simulación se realiza mediante raytracing hace que el diseñador de efectos especiales deba esperar varios segundos o minutos entre cada iteración para poder llegar al resultado buscado.

Este tipo de inconvenientes puede ser parcialmente mitigado con una CPU lo suficientemente potente como para reducir los tiempos de espera entre simulación y renderización. Aún así, los tiempos de espera serán cada vez mayores a medida que la complejidad de la simulación (y visualización de la misma) vaya aumentando. Esto limita la experimentación por parte del diseñador, haciendo que éste tienda a ignorar posibles variaciones interesantes debido al tiempo que tomaría realizar las debidas pruebas.

La industria de los videojuegos no siempre contó con la posibilidad de realizar simulaciones de fluidos, debido principalmente a la falta de poder de procesamiento necesaria para correr una simulación que provea resultados aceptables dentro de las restricciones de tiempo junto al resto de las tareas que se dan lugar en el *game loop*[11], como la simulación de la física, inteligencia artificial, procesamiento del input del jugador, renderizado de la escena, etc. Por los motivos anteriormente mencionados, una implementación en una CPU no es viable en los videojuegos, y las GPUs originalmente no contaban con el poder de procesamiento necesario para encargarse de tales tareas. No obstante modelos suficientemente simples fueron presentados, los cuales eran capaces de correr a framerates interactivos, posibilitando su uso en los videojuegos **[8]**.

Con la progresiva evolución de las GPUs, surgieron nuevas posibilidades a raíz del incremento en su flexibilidad y poder de procesamiento, las cuales permitieron la implementación de una simulación de fluidos enteramente en la GPU **[42][43]**. Sin embargo, en términos de flexibilidad no se la puede comparar a la CPU y su renderizado utilizando raytracing, pero aún así es posible realizar simulaciones y visualizaciones de buena calidad manteniendo un framerate acorde al de una aplicación en tiempo real.

Esto creó la posibilidad de que se puedan desarrollar herramientas que permitan al diseñador de efectos especiales (en la industria cinematográfica) realizar una aproximación del efecto buscado utilizando la GPU **[7]**. Aprovechándo el renderizado en tiempo real tendrá una mayor libertad para experimentar, y una vez conforme, volverá a la utilización de

[11] El *game loop* es una función que se ejecuta una vez por frame, y es el núcleo de toda aplicación interactiva.

la CPU para refinar el efecto buscado valiéndose de los parámetros de la simulación previamente utilizados en la GPU. Esto reduce enormemente el tiempo que toma la producción de efectos especiales impulsados por la simulación de fluidos.

En el área científica, también es posible utilizar la GPU para la simulación de fluidos en respuesta a catástrofes naturales, como por ejemplo tsunamis, realizando una simulación del desplazamiento de agua provocado por un terremoto en el océano y visualizando rápidamente las principales zonas que serán afectadas por el mismo. Permitiendo así una evacuación mas rápida y eficiente **[9][44]**.

Finalmente, a diferencia de las películas, los videojuegos utilizarán la GPU para la simulación de fluidos tanto durante su creación como durante la ejecución del videojuego propiamente dicho, lo cual hace que se deba buscar un balance entre calidad y *performance* para obtener una experiencia fluida.

Como se puede observar, al menos todas las áreas mencionadas pueden beneficiarse de una herramienta que permita realizar una simulación en tiempo real. Por lo tanto los conceptos involucrados en el desarrollo de dicha herramienta deben ser estudiados para poder cubrir exitosamente las necesidades del área en la que se lo requiera.

5. OBJETIVO

El objetivo de esta tesina será el de presentar todos los conceptos teóricos necesarios para poder implementar una simulación de fluidos en tiempo real. Para esto, se partirá de las ecuaciones de Navier-Stokes, que luego serán discretizadas para poder ser implementadas en la GPU. Dada la extensión del tema, se ha decidido acotar el tipo de fluido al de los fluidos gaseosos, realizando simplificaciones en donde sea posible.

Junto con la explicación teórica se desarrollará un programa que ponga en práctica todos los conceptos presentados, proveyendo de código en donde sea necesario. Dicho programa realizará una simulación de humo en tiempo real, contemplando solamente la densidad del humo y las fuerzas de la flotabilidad dada la temperatura.

Para el desarrollo de la aplicación se utilizará Unity3D como framework, dada sus capacidades para el prototipado rápido, buena documentación e integración con las APIs de DirectCompute y Direct3D. Dichas APIs irán a utilizarse en la GPU para realizar la simulación y visualización del fluido respectivamente.

Se ha decidido utilizar la GPU debido a que provee un buen balance entre interactividad y calidad gráfica **[42]**, el cual es adecuado para los tipos de aplicaciones previamente mencionadas. Finalmente, se presentarán todos los detalles pertinentes con respecto a la arquitectura de la GPU en donde se ejecutará el programa, y por último se hará mención del funcionamiento del pipeline gráfico que será necesario para poder visualizar los resultados de la simulación.

6. ESTADO DEL ARTE

Los primeros modelos para la simulación de fluidos no estaban físicamente basados en el comportamiento del mismo, en su lugar se concentraban en la apariencia visual siendo creados a partir de simples primitivas, los cuales luego permitieron a los animadores crear sistemas basados en partículas.

Con el paso del tiempo surgieron modelos basados en las ecuaciones de navier-stokes, siendo Kajiya y Von Herzen los primeros en utilizarlas en el área de Computer Graphics (CG) **[10]**. Dichos modelos tenían la particularidad de ser inestables (debido a su implementación) si se utilizaba un time step (intervalo de tiempo entre cada "paso" de la simulación) muy grande. Esta inestabilidad numérica lleva al colapso de la simulación luego de un tiempo debido a la acumulación de errores, lo cual llevaba al reinicio de la simulación utilizando un *time step* menor.

En 1999 Jos Stam **[1]** propuso una solución a la ecuación de Navier-Stokes resolviendo el problema de la inestabilidad, utilizando una combinación de advección semi-lagrangiana y métodos implícitos. Gracias a esto se pudo correr la simulación utilizando *time steps* mayores, permitiendo al usuario interactuar con la simulación en tiempo real (en computadoras lo suficientemente potentes). Sin embargo, esta técnica sufría el problema de la "disipación numérica", el cual hacía que el flujo se disipe demasiado rápido a comparación de los experimentos realizados.

En solución a esto, en 2001 Fedkiw et. al. **[11]** introdujeron el "vorticity confinement" el cual consiste en "inyectar" en el fluido la energía perdida debido a la disipación numérica, manteniéndolo así fluyendo durante un mayor tiempo. Junto a esto también propusieron una técnica para la simulación de humo en aplicaciones de computación gráfica, utilizando las ecuaciones de Euler para flujos incompresibles.

En el 2000 Yngve et. al. **[12]** propuso utilizar la versión de las ecuaciones de fluidos compresibles para modelar explosiones. Si bien las ecuaciones para flujos compresibles son útiles para modelar ondas de choque y otros fenómenos similares, éstas imponen un *time step* muy estricto asociado a las ondas acústicas. Normalmente se suele evitar esta restricción utilizando las ecuaciones incompresibles siempre que sea posible.

En el 2003 Goodnight et al. **[2]** fue uno de los primeros en proponer el uso de la GPU para resolver las restricciones de borde (o boundary value problem) en las ecuaciones diferenciales, como aquellas utilizadas en la dinámica de fluidos. Para la implementación utilizaron el método de múltiples grillas (multigrid method), demostrando incrementos de velocidad de hasta 15 veces con respecto a técnicas implementadas en la CPU.

En el 2010 Cohen et al.**[13]** propusieron un sistema de simulación de fluidos interactiva (en la GPU) utilizando una grilla euleriana (eulerian grid) móvil, permitiendo que el fluido pueda desplazarse por el espacio sin estar restringido a una posición fija del tipo "fluid in a box". A medida que las partículas del fluido dejan el dominio de la simulación, se hace una transición a un sistema tradicional de partículas para ocultar el momento en el cual la simulación de fluidos termina.

En el 2016 Tompson et al. **[14]** propuso utilizar métodos de *machine learning* para optimizar la etapa computacionalmente más intensiva en la ecuaciones de Euler, la presión, utilizando una convolutional neural network (ConvNet). Para resolver la presión, se debe satisfacer la restricción de incompresibilidad, y para esto se suele utilizar la ecuación discreta de Poisson. El uso de dicha ecuación resulta en un algoritmo que deberá realizar varias iteraciones para llegar a la solución buscada, y en el ámbito de las aplicaciones en tiempo real muchas veces se debe truncar el número de iteraciones necesarias antes de alcanzar un buen resultado para evitar invertir muchos recursos en la simulación. El artículo propone utilizar una ConvNet para derivar un mecanismo de inferencia aproximado que se aprovecha de los datos estadísticos de los fluidos, estableciendo el problema como una tarea de regresión.

7. SIMULACIÓN DE FLUIDOS: TEORÍA

En la sección 7.1 se presentará la ecuación de Navier-Stokes para flujos incompresibles, y se analizará cada una de sus partes. Luego, se simplificará la ecuación de Navier-Stokes resultando en la ecuación de Euler para flujos incompresibles, la cual se utilizará para el modelado de gases **[15]**. En la sección 7.2 se discutirá la forma de discretizar la ecuación de Euler, presentando también la técnica de advección que permitirá realizar una simulación estable.

7.1 Introducción

Como se mencionó anteriormente, encontrar la forma de simular fluidos de forma realista es equivalente a desarrollar un buen *fluid solver* para las ecuaciones de Navier-Stokes. Sin embargo estas ecuaciones son particularmente complejas de resolver dada su propiedad de no-linealidad, lo cual lleva a tener que realizar simplificaciones apropiadas al ámbito en el que se las requiera utilizar, como por ejemplo asumir que una de las variables es constante con respecto al tiempo.

Los fluidos pueden ser categorizados en dos tipos: compresibles e incompresibles. Los fluidos son incompresibles si no presentan un cambio en su densidad (y por lo tanto en su volumen) a causa de un cambio en la presión, por ejemplo provocado por fuerzas externas. La incompresibilidad es una propiedad que posee el fluido.

En la realidad todos los fluidos son compresibles, y la cantidad de presión necesaria para comprimirlos depende del fluido en cuestión, determinado por un índice de compresibilidad que todos los fluidos poseen. **[16]**

Es necesario hacer una distinción entre fluido incompresible y flujo incompresible. Un flujo es incompresible cuando la densidad de una "parte" infinitesimal del fluido se mantiene constante a medida que ésta es transportada por la velocidad del fluido. Cada una de las partes puede tener una cantidad de densidad distinta, pero las densidades serán constantes a lo largo de su movimiento.

En el caso del aire, su compresibilidad a bajas velocidades es insignificante (y por lo tanto su densidad será constante), sin embargo la compresión del aire aumenta a medida que se aproxima y excede la velocidad del sonido.

Dado que en la implementación de esta tesina la velocidad del fluido estará por debajo de la velocidad del sonido, también es posible asumir que el flujo es incompresible, simplificando el problema y permitiendo así utilizar las ecuaciones de Navier-Stokes para flujos incompresibles.[12]

[12] En la práctica es común asumir que un flujo es incompresible dado que esto permite simplificar el problema, esto es gracias a que generalmente las velocidades de los fluidos que se intentan simular no se mueven a la velocidad del sonido. Si se quisiese simular el comportamiento del aire ante una explosión, debería modelarse con un flujo compresible.

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{u} + \vec{F}$$

Ecuación 1

$$\nabla \cdot \vec{u} = 0$$

Ecuación 2

La ecuación 1 y 2 componen lo que se conoce como ecuaciones de Navier-Stokes. La ecuación 1 es llamada ecuación del momentum, la ecuación 2 es la condición de incompresibilidad. Antes de comenzar a explicar el significado y derivación de cada término, se presentará una visión intuitiva de cómo funcionan los fluidos, para luego asociar los conceptos a las ecuaciones.

Condición de incompresibilidad

En lo que respecta a las ecuaciones de Navier-Stokes, fue Leonhard Euler el primero en formalizar el movimiento de los fluidos en un conjunto de ecuaciones. Su primera contribución fue la condición de incompresibilidad para el flujo de un fluido, y la segunda fue derivar una ecuación para la velocidad de un fluido "ideal" asumiendo que éste es incompresible.

Euler consideraba un fluido ideal a aquellos que están en constante movimiento, a causa de la falta de viscosidad que reduzca la velocidad con el paso del tiempo. Como se mencionó anteriormente, un fluido con flujo incompresible se lo puede ver como aquel que puede cambiar de forma, pero siempre mantendrá su volumen.

Antes de continuar, es necesario introducir el concepto de velocity field. Es posible observar que algunos fluidos (como el aire) no pueden ser vistos, pero pueden ser percibidos mediante los efectos que produce en los elementos que se suspenden en él (como por ejemplo partículas de polvo). Los fisicos deciden representar el movimiento de dichos fluidos utilizando un campo vectorial, el cual consiste en asignar un vector a cada punto infinitesimal en el espacio en donde dicho vector indica la dirección y magnitud de la velocidad.

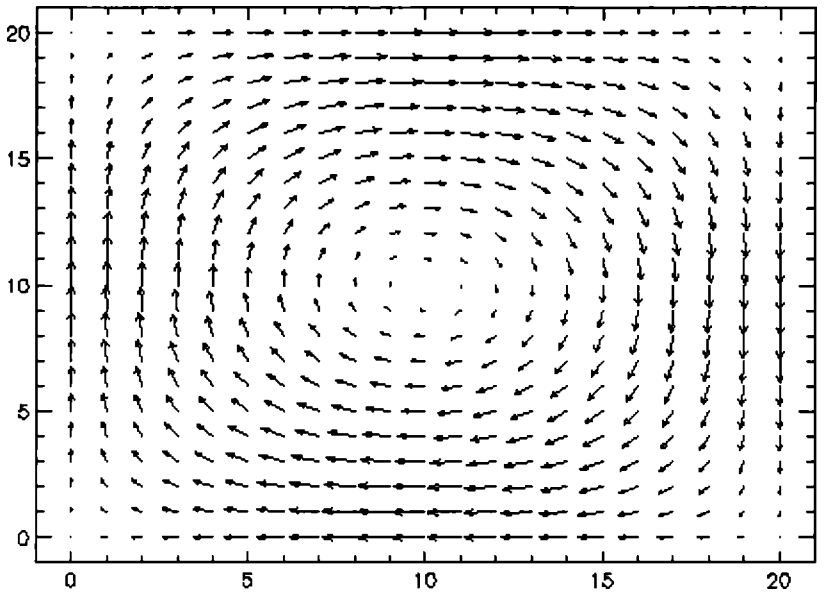


Figura 7: Ejemplo de un velocity field.

Dado que asignar un vector a cada punto en el espacio resulta imposible, se suele particionar el espacio en secciones (por ejemplo una grilla de 20x20 elementos) a las cuales se les asigna un único vector.

Si se considera un fluido moviéndose a través de dicha grilla, es posible observar que en cada celda hay un flujo de entrada y un flujo de salida con respecto a sus celdas vecinas (horizontal y verticalmente). En otras palabras, se produce un intercambio de energía de algún tipo.

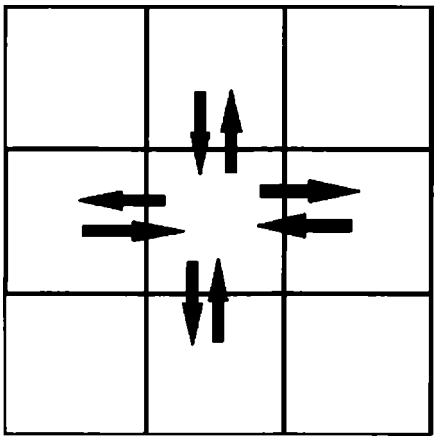


Figura 8: Flujo entrante y saliente de una celda.

En la figura 8 se puede observar el flujo que entra y sale de la celda ubicada en el centro. La condición de incompresibilidad propuesta por Euler es equivalente a decir que la cantidad de flujo entrante, debe ser igual a la cantidad de flujo saliente.

En este contexto, cuando se habla de flujo (flux) se refiere al transporte de alguna sustancia o propiedad, y dicho transporte está determinado por un vector representando la dirección en la que se produce, y la cantidad.

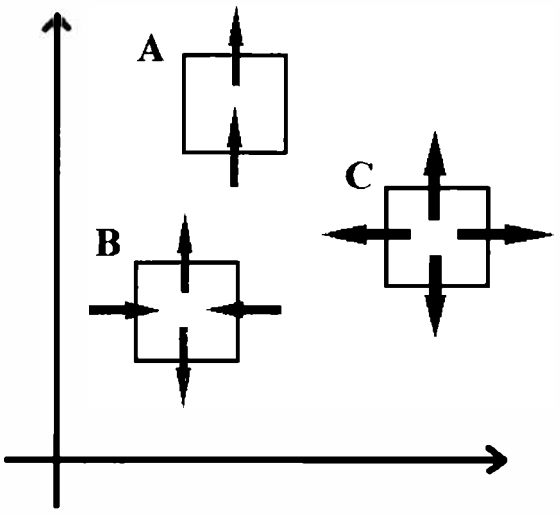


Figura 9: Flujo entrante y saliente para distintos puntos del fluido.

En la figura 9 se pueden observar partes de un fluido tomadas en distintos puntos, junto con el flujo de entrada y salida correspondiente a cada parte, denotado con una flecha. Suponiendo que todas las flechas representan la misma cantidad de flujo constante, es decir todos los vectores tienen la misma magnitud, entonces en las figuras A y B el flujo que fluye hacia la celda es igual a la cantidad de flujo que fluye hacia afuera de la celda.

En el punto C se puede ver que todas las flechas apuntan hacia afuera, lo cual hace que esa celda actúe como una fuente de flujo, haciendo que el flujo se expanda y por lo tanto no cumplirá con la condición de incompresibilidad.

Para dar un ejemplo más concreto, en la siguiente figura se considerará el in-flux (flujo entrante) con una magnitud de 10, y un out-flux (flujo saliente) con una magnitud de 20.

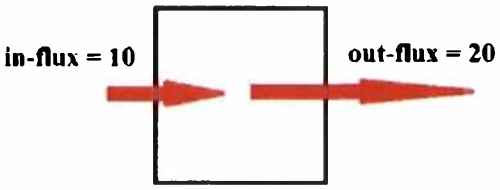


Figura 10: in-flux y out-flux.

Continuando con el ejemplo, si se supone que la densidad del fluido está siendo transportada por el flujo, puede observarse que en un momento dado sale desde la celda de fluido una mayor cantidad de densidad que la que entra. Intuitivamente se puede deducir que en ese punto se están generando 10 unidades extra de densidad.

Dicho esto, la condición de incompresibilidad podría ser definida de la siguiente forma:

$$flujo_{entrada} - flujo_{salida} = 0$$

(asumiendo que el flujo es un número positivo)

Expresado de forma más rigurosa, el teorema de la divergencia dice que para una región R del fluido, teniendo un borde C con un vector normal $\vec{n}$, y un vector field representando la velocidad del fluido para toda la región R, entonces la suma del producto escalar entre el flujo de velocidad **F** y el vector normal $\vec{n}$ para todo el borde C, es lo mismo que integrar la divergencia de la velocidad F para toda la región. Es decir:

$$\int_C \vec{F} \cdot \vec{n} = \iint_R \nabla \cdot \vec{F}$$

Ecuación 3: Teorema de la divergencia en dos dimensiones.

Si para cada uno de los puntos en el borde de la región se evalúa el producto escalar entre la velocidad que cruza ese borde, y el vector normal de dicho borde, es posible integrar el resultado a lo largo de todo el borde para así obtener el flux total de velocidad.

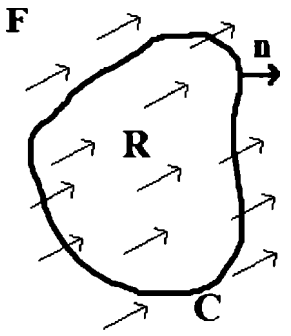


Figura 11: Flujo sin divergencia.

La condición de incompresibilidad indica que la densidad del fluido debe ser constante, esto quiere decir que el flujo entrante al fluido debe ser igual al flujo saliente. Por lo tanto para **todo el borde** de la región de fluido, se debe cumplir que:

$$\nabla \cdot \vec{F} = 0$$

Es decir, el flujo en el borde debe ser tangencial al vector normal, para todo el borde. El teorema de la divergencia permite tomar la divergencia de la velocidad en toda la región en lugar de evaluar el producto escalar en el borde, lo cual simplifica los cálculos. Si se aplica esta condición para todo el fluido, y no solo una región determinada, se obtiene la condición de incompresibilidad.

En el ejemplo de la Figura 11 si se calcula la divergencia para toda la región R, el resultado será 0, y es posible observar que a la región entra y sale la misma cantidad de flujo.

En el ejemplo de la Figura 12, la divergencia será positiva dado que el flujo saliente es mayor al entrante:

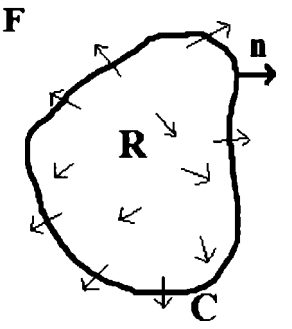


Figura 12: Flujo con divergencia

En conclusión, la condición de incompresibilidad en las ecuaciones de Navier-Stokes está representada por la divergencia de la velocidad, de forma tal que:

$$\nabla \cdot \vec{u} = 0$$

con $\vec{u}$ representando el vector de la velocidad.

Presión

La presión se genera cuando las partículas colisionan entre sí, provocando el desplazamiento de otras partículas y propagando el movimiento a lo largo de todo el fluido. Las partículas se desplazan de zonas de mayor presión a zonas de menor región, y este comportamiento está representado por el término de la presión en las ecuaciones:

$$-\frac{1}{\rho}\nabla p$$

Nótese que el gradiente apunta hacia la zona de mayor crecimiento. Por lo tanto, se toma la dirección contraria para ir a la zona de menor presión.

Resolver las ecuaciones de Navier-Stokes involucra calcular tres términos para actualizar la velocidad en cada time step: advección, difusión, y aplicación de fuerzas externas (por ejemplo gravedad)[13].

El resultado de estos cálculos es una nueva velocidad cuya divergencia es distinta de cero. Previamente se mencionó que para satisfacer la condición de incompresibilidad, la velocidad no debe poseer divergencia.

Para esto se utilizará la descomposición de Helmholtz-Hodge la cual indica que todo campo vectorial puede ser descompuesto en la suma de un campo incompresible y el gradiente de un campo escalar. **[17]**

[13] La advección y difusión se explicarán en las siguientes secciones.

Este campo escalar es la presión **p** del fluido, y por lo tanto representará la cantidad de fuerza necesaria para mantener al campo de velocidad libre de divergencia. En la sección de la discretización de las ecuaciones se proveerán más detalles acerca del cálculo de la presión.

En conclusión, la presión será utilizada para mantener constante al volumen del fluido asegurándose de que la velocidad se mantenga sin divergencia, respetando así la condición de incompresibilidad.

Advección

La velocidad de un fluido indica cómo las cosas inmersas en él se mueven con respecto al tiempo, y encontrar una ecuación que describa su cambio es fundamental para la simulación del comportamiento de los fluidos.

La segunda gran contribución de Euler fue cómo la velocidad de un fluido ideal evoluciona a lo largo del tiempo, y una de sus propiedades más importantes es la no-linealidad.

Antes de introducir estos conceptos se debe hacer mención de los distintos puntos de vista que se pueden utilizar para medir el cambio de la velocidad con respecto al tiempo. A continuación se presenta un simple ejemplo que intenta explicar dichos conceptos.

Se supondrá que se posee una piscina con un nadador en el centro, dicha pileta se encuentra al aire libre, y se desea medir el cambio de la temperatura del agua con el nadador estando completamente quieto. A medida que pasa el tiempo, el sol irá calentando el agua y el nadador podrá percibir un leve cambio en la temperatura a causa de esto. El resultado será el cambio de temperatura en una posición fija a lo largo del tiempo. Este es un tipo de medición utilizando el punto de vista Euleriano, el cual consiste en realizar mediciones en puntos fijos sobre un área, sin cambiar de posición.

Luego, se supondrá que la pileta se encuentra bajo techo en un ambiente cerrado, en un extremo de la pileta se posee un calefactor que calienta el agua gradualmente hasta llegar a un punto de estabilidad, también llamado steady state, en el cual la temperatura no cambiará con respecto al tiempo. Nuevamente se desea medir la temperatura pero esta vez haciendo que el nadador se mueva a lo largo del tiempo para tomar las mediciones. En este caso si el nadador se acerca a la fuente de calefacción, notará un aumento en la temperatura, y si se aleja notará un decremento de la misma.

El resultado será que el nadador detecta nuevamente un cambio en la temperatura con respecto al tiempo, pero esta vez dicho cambio se debe al movimiento. Si el nadador se queda quieto en un punto no notará ningún cambio. Esta forma de realizar las mediciones utiliza el punto de vista lagrangiano, en el cual los resultados dependen de una posición y un momento en el tiempo.

Una función que mide el cambio en la temperatura desde el punto de vista Euleriano puede ser definida de la siguiente forma:

$$\frac{\partial T}{\partial t}(t)$$

el parámetro t representa el tiempo, y dado que se está utilizando el punto de vista Euleriano, la posición es constante. Por lo tanto, solo hace falta saber un momento en el tiempo para determinar el cambio en la temperatura. Una función que mide el cambio de la temperatura desde el punto de vista lagrangiano, puede ser definida de la siguiente forma:

$$\frac{\partial T}{\partial t}(X(t))$$

El parámetro **X** representa la posición, el cual a su vez es una función que depende del tiempo. Esto se debe a que en el punto de vista lagrangiano el cambio de temperatura está determinado por la posición del nadador en un momento dado.

Es posible combinar ambos puntos de vista para conseguir lo que se conoce como derivada total (o derivada material), para conseguir así el cambio con respecto a una posición fija, y con respecto al movimiento de una partícula de fluido que a su vez depende del tiempo. Definiéndose de la siguiente forma:

$$\frac{\partial T}{\partial t}(t,\ X(t))$$

Se puede resolver esta función utilizando la regla de la cadena, asumiendo que la posición es un vector en 2 dimensiones, tal que[14] X = (x,y) :

$$\frac{\partial T}{\partial t}(t,\ x(t),\ y(t)) = \frac{\partial T}{\partial t} + \frac{\partial T}{\partial x}\frac{dx}{dt} + \frac{\partial T}{\partial y}\frac{dy}{dt}$$

Luego, se puede definir a la velocidad como V = (u,v), en donde:

$$u = \frac{dx}{dt}$$
$$v = \frac{dy}{dt}$$

Luego:

$$\frac{\partial T}{\partial t}(t,\ x(t),\ y(t)) = \frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}$$

Y finalmente, ésta última ecuación puede ser reducida utilizando el operador nabla como el gradiente de la temperatura con respecto a *x* e *y*:

$$\frac{DT}{Dt} = \frac{\partial T}{\partial t} + (V \cdot \nabla)T$$

[14] La variable X representa un vector en dos dimensiones, siendo **x** e **y** los componentes de dicho vector.

Esta es la forma de derivada total, nótese la D mayúscula. El primer término del lado derecho mide la cantidad de cambio de temperatura en un punto fijo con respecto al tiempo (local rate of change), y el segundo término mide la cantidad de cambio de temperatura con respecto al movimiento (advective rate of change, también llamado término advectivo).

El resultado de la derivada total es un número escalar, pero también es posible tomar la derivada de un vector de N dimensiones. Para esto se debe calcular el resultado por cada uno de los componentes del vector por separado, es decir, para un vector U=(*x,y)* entonces:

$$\frac{DU_x}{Dt} = \frac{\partial U_x}{\partial t} + (V \cdot \nabla)U_x$$
$$\frac{DU_y}{Dt} = \frac{\partial U_y}{\partial t} + (V \cdot \nabla)U_y$$

Utilizando el concepto de derivada total es posible ver cómo los distintos atributos del fluido (no solo la temperatura) evolucionan a lo largo del tiempo dentro del fluido.

Al comienzo de esta sección se mencionó que la velocidad del fluido indica cómo las cosas se mueven en él, y encontrar una ecuación que describa dicha velocidad es fundamental para la simulación del fluido. Euler descubrió que el cambio en la velocidad depende de la velocidad misma, es decir:

$$\frac{D\vec{u}}{Dt} \equiv \frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u}$$

O expresado de otra forma:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u}$$

Por esto último se dice que la velocidad no es lineal, lo cual agrega una mayor dificultad al momento de resolver las ecuaciones.

Durante la implementación de las ecuaciones uno está interesado en calcular el cambio en la velocidad en un punto dado (es decir utilizando un punto de vista Euleriano) y por lo tanto se utilizará la ecuación en la última forma descrita.

Fuerzas externas

Las fuerzas externas (indicado con una **F** en las ecuaciones) representa una fuerza en forma de vector que afecta a la velocidad de un fluido, como por ejemplo la gravedad, o incluso una fuerza generada por el usuario por medio de la interacción con la aplicación.

Esta fuerza puede ser continua (como en el caso de la gravedad) o temporal (en el caso de la interacción del usuario), esto quiere decir que pueden haber momentos en los cuales el fluido no reciba fuerzas externas haciendo que eventualmente se detenga a causa de la viscosidad.

Viscosidad

En algunas situaciones, las fuerzas de la viscosidad son extremadamente importantes, por ejemplo para simular substancias como la miel o flujos de fluidos a pequeña escala.

Sin embargo en muchos otros casos la viscosidad juega un rol menor, como en los gases en donde la viscosidad es insignificante, y es por esto que suele ser ignorada en las ecuaciones. Sin embargo muchos de los métodos numéricos utilizados suelen introducir errores de precisión los cuales pueden ser reinterpretados como viscosidad, por lo que incluso sin modelar la viscosidad, se obtiene un comportamiento muy similar.

Uno de los desafíos más grandes durante la simulación de fluidos es intentar combatir este tipo de errores para obtener un resultado más preciso **[18]**.

Las ecuaciones de Navier-Stokes sin tomar en cuenta la viscosidad son llamadas ecuaciones de Euler[15], las cuales se pueden observar a continuación, tomando en consideración una densidad constante igual a 1 [16]:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \vec{F}$$

Ecuación 5

$$\nabla \cdot \vec{u} = 0$$

Ecuación 6

$\vec{u} = velocidad$

$t = tiempo$

$p = presión$

$\vec{F} = fuerzas\ externas$

La Ecuación 5 (ecuación del momentum) puede ser reescrita utilizando la derivada material, como se muestra a continuación:

$$\frac{D\vec{u}}{Dt} = -\nabla p + \vec{F}$$

Se puede apreciar la similitud que posee con la ecuación de la segunda ley de Newton:

$$\vec{F} = m\vec{a}$$

$$\vec{a} = \vec{F}/m$$

Por lo tanto las ecuaciones de Euler (o de Navier-Stokes) no son más que la segunda ley de Newton actuando sobre el fluido.

---

[15] Y los fluidos sin viscosidad son llamados fluidos inviscidos (inviscid fluids).

[16] Normalmente el gradiente de la presión va acompañado por la multiplicación de *1/densidad*, al hacer que la densidad sea igual a 1 es posible evitar agregar la multiplicación en la ecuación.

Fluidos gaseosos

Las ecuaciones mencionadas anteriormente indican cómo evoluciona la velocidad de un fluido (en este caso el aire) a lo largo del tiempo, sin embargo para modelar gases como el humo o vapor es necesario disponer de atributos esenciales, estos son la temperatura y la densidad.

En este caso cuando se habla de la densidad, se refiere a la densidad del gas, y no del fluido. Se puede ver al humo como un conjunto de partículas pequeñas de carbón (o de agua en el caso del vapor) las cuales son transportadas por un fluido como el aire, y por lo tanto se debe distinguir entre la densidad del fluido y la densidad del gas. La densidad del fluido se asume que es constante, mientras que la densidad del gas puede variar.

Para reflejar el hecho de que tanto la temperatura como la densidad son transportados por la velocidad del fluido, se utilizarán las siguientes ecuaciones:

$$\frac{\partial T}{\partial t} = -(\vec{u} \cdot \nabla)T$$

Ecuación 7

$$\frac{\partial \rho}{\partial t} = -(\vec{u} \cdot \nabla)\rho$$

Ecuación 8

en donde T representa la temperatura y ρ (rho) representa la densidad del gas.

Tanto la temperatura como la densidad, afectan a la velocidad del fluido. El humo con alta densidad tiende a caer debido a la gravedad, mientras que gases calientes tienden a elevarse debido a las fuerzas de la flotabilidad.

A continuación se presentará un modelo simple propuesto por Fedkiw et. al. **[11]** para integrar este comportamiento a las fuerzas externas de la ecuación del momentum:

$$f_{buoy} = -\alpha\rho y + \beta(T - T_{amb})\, y$$

Ecuación 9

en donde:

y = vector del eje vertical <0,1,0>

ρ = densidad del humo

T = temperatura del humo

T_{amb} = temperatura ambiente

α y β = son dos constantes positivas con las unidades adecuadas de forma tal que el resultado sea físicamente plausible.

Nótese que si la densidad es 0 y la temperatura es igual a la temperatura ambiente, el resultado de la fuerza será 0.

Finalmente, la ecuación de la velocidad del fluido quedará de la siguiente forma (utilizando la derivada total, para denotar su simplicidad)[17]:

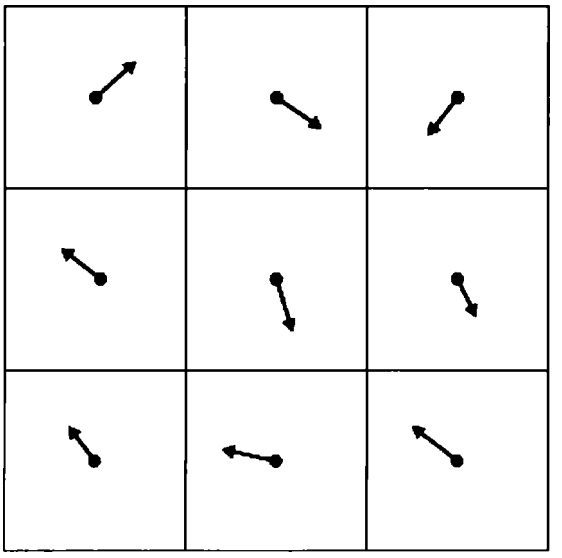
$$\frac{D\vec{u}}{Dt} = -\nabla p + \vec{f_{buoy}}$$

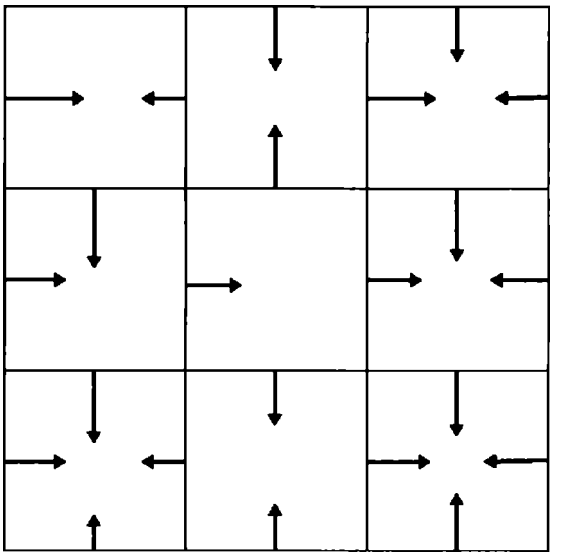
Ecuación 10

7.2 Resolviendo las ecuaciones

Antes de poder implementar las ecuaciones, es necesario realizar una discretización espacial y temporal. Para la discretización espacial se utilizarán varias grillas que almacenarán la velocidad, presión, densidad y temperatura; y se asume que los dichos valores se encuentran en el centro de cada celda. Este método de discretización se lo conoce como cell-centered discretization ilustrado en la Figura 13.

Otro método de discretización espacial muy utilizado consiste en representar los atributos del fluido (velocidad, presión, etc) en los bordes de cada celda. Dicho método es conocido como staggered grid, y provee una mayor precisión en los cálculos, además de reducir las oscilaciones producidas al aplicar fuerzas como la flotabilidad a diferencia de una cell-centered grid.





[17] Es importante recordar que la ecuación completa de Navier-Stokes toma en cuenta la viscosidad, la cual en este caso por los motivos anteriormente mencionados, se decidió dejarla de lado.

Figura 13: A la izquierda cell-centered grid, y a la derecha staggered grid.

Dicho esto, por cuestiones de simplicidad se utilizará el método de cell-centered grid, ya que permite realizar una implementación más simple gracias la simplicidad del método.

Para la discretización temporal, se calcularán las operaciones diferenciales utilizando los métodos de diferencia finita (tomando en cuenta el hecho de que los valores de la velocidad, presión, temperatura y densidad estarán ubicados en el centro de cada celda de la grilla). En particular, se utilizará el método de diferencia finita central, dada por la siguiente función:

$$\delta_h f(x) = \frac{f(x+h)-f(x-h)}{2h}$$

En el contexto de la grilla, *h* representa la distancia entre celdas, la cual será equivalente al valor 1(elegido arbitrariamente). Por consiguiente es posible reescribir la función de la siguiente forma, esta vez utilizando una notación que refleje el uso de la grilla en dos dimensiones. Los subíndices denotan la posición(o el índice) de la celda en dicha grilla:

$$\nabla f_{i,j} = < \frac{f_{i+1,j}-f_{i-1,j}}{2}, \frac{f_{i,j+1}-f_{i,j-1}}{2} >$$

Aquí se utiliza el método de diferencia finita central para calcular el gradiente de la función en la posición *i,j* (equivalentes al componente horizontal y vertical respectivamente), dando como resultado un vector. Nótese que la función f es evaluada sobre un campo escalar de dos dimensiones.

El cálculo de la divergencia se realiza de forma similar:

$$\nabla \cdot \vec{f}_{i,j} = \frac{f_{i+1,j}-f_{i-1,j}}{2} + \frac{f_{i,j+1}-f_{i,j-1}}{2}$$

Ecuación 11

En este caso f es evaluada sobre un campo vectorial. En el primer término del lado derecho solo se toma el componente horizontal, mientras que en el segundo término del lado derecho se toma el componente vertical de cada vector, de la siguiente forma:

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

en donde

$$\vec{u} = < u, v >$$

De esta forma es posible calcular la cantidad de flujo horizontalmente y verticalmente, para luego sumar los resultados y obtener la divergencia.

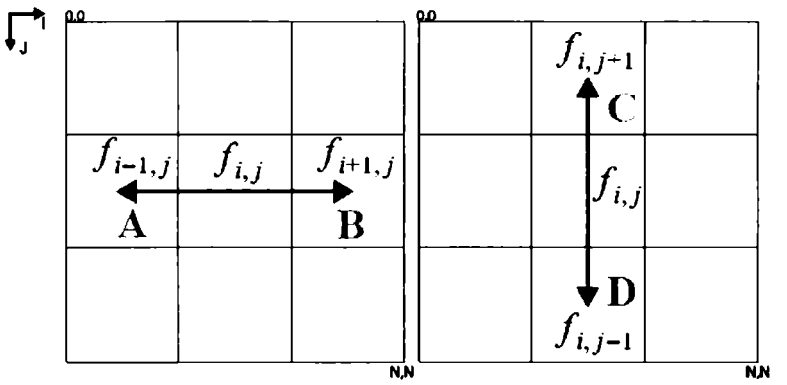


Figura 14: Cálculo de la divergencia

En lo que resta del capítulo se presentarán las técnicas necesarias para discretizar temporalmente (con respecto al tiempo) cada uno de los términos de las ecuaciones, de las cuales lo que se quiere calcular es el cambio de la velocidad, temperatura y densidad en un punto fijo (representado por una celda en la grilla). Las ecuaciones finales son las siguientes:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \vec{f_{buoy}}$$

Ecuación 12

$$\frac{\partial T}{\partial t} = -(\vec{u} \cdot \nabla)T$$

Ecuación 13

$$\frac{\partial \rho}{\partial t} = -(\vec{u} \cdot \nabla)\rho$$

Ecuación 14

A continuación se describirá la forma en la que se resolverá la ecuación del momentum (12).

El objetivo es partir de una velocidad vectorial u^t, la cual se asume que no posee divergencia, y llegar a una nueva velocidad u^{t+1} (en el siguiente instante de tiempo), la cual de acuerdo a la condición de incompresibilidad, tampoco poseerá divergencia.

Se debe tener en cuenta que desde este momento en adelante se utilizará la notación de las potencias (u^t) para denotar una cantidad cualquiera (en este caso la velocidad) en un instante de tiempo **t**, y por lo tanto no se lo debe confundir con la notación de las potencias. En los casos que se utilicen potencias, se hará una mención explícita del mismo.

Operator splitting

Como con cualquier algoritmo, será necesario dividir la ecuación del momentum en pasos simples de resolver, utilizando el método descrito por Jos Stam en su publicación Stable Fluids **[19]**.

Primero se transformará la ecuación de la velocidad en una forma más amigable para poder resolverla numéricamente. Partiendo de la siguiente ecuación:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \vec{f_{buoy}}$$

Será descompuesta en cada uno de los componentes del vector de velocidad $\vec{u}$ para resolver por separado (como se mencionó en la sección de advección). Por lo tanto, para el vector de velocidad:

$$\vec{u} = (u, v)$$

se tendrán las siguientes ecuaciones:

$$\frac{\partial u}{\partial t} = -(\vec{u} \cdot \nabla)u - \nabla p + \vec{f_{buoy}}$$
$$\frac{\partial v}{\partial t} = -(\vec{u} \cdot \nabla)v - \nabla p + \vec{f_{buoy}}$$

Como se puede observar, se cuenta con tres incógnitas a resolver: ***u***, ***v***, y ***p***. Sin embargo aún no es claro como pueden ser resueltas.

Tal y como menciona Stam en su publicación, resolver la ecuación de Navier-Stokes involucra realizar el cómputo de la advección, difusión (la cual en esta tesina no es utilizada) y la presión para poder actualizar la velocidad en cada timestep. El resultado de estos pasos es un campo vectorial de velocidad **w** con divergencia. Sin embargo, la restricción de incompresibilidad indica que la velocidad no debe poseer divergencia.

En 1967 Alexandre Chorin propuso un método**[20]** para resolver las ecuaciones de Navier-Stokes. En él, Chorin utiliza el teorema de la descomposición de Helmholtz-Hodge para calcular una velocidad intermedia (con divergencia) y luego utilizando la presión le aplica una proyección (corrección) a la velocidad para llegar a un resultado sin divergencia.
A continuación se presenta el teorema de Helmholtz-Hodge, y el método de proyección utilizado por Chorin para resolver las ecuaciones de Navier-Stokes.

Teorema de la descomposición de Helmholtz-Hodge

El teorema indica que cualquier campo vectorial puede ser descompuesto en la suma de un campo vectorial solenoidal (sin divergencia) y un campo vectorial irrotacional (sin rotaciones [18]). Es decir:

$$u = u_{sol} + u_{irrot}$$

Siendo **u** un campo vectorial cualquiera.

Luego, una de las identidades del cálculo vectorial **[21]** indica que el curl (es decir la cantidad de rotación que posee un vector en un punto dado) del gradiente de un campo escalar cualquiera, que posea una segunda derivada, da como resultado 0. Es decir, para un campo escalar ϕ cualquiera que sea doblemente diferenciable, tenemos que:

$$\nabla \times \nabla \phi = 0$$

Por lo tanto, es posible utilizar esta identidad para reemplazarla en la ecuación de descomposición, resultando en lo siguiente:

$$u = u_{sol} + \nabla \phi$$

Aquí, **u** representa nuestra velocidad intermedia, por lo tanto utilizando esta ecuación es posible despejar $\mathbf{u_{sol}}$, llegando a la velocidad deseada sin divergencia:

$$u_{sol} = u - \nabla \phi$$

Como se puede observar, el cálculo de la velocidad sin divergencia es obtenido primero computando una velocidad intermedia **u**, y luego se le debe aplicar una "corrección" restandole el gradiente de un campo escalar.

Sin embargo, aún queda una incógnita por resolver, y ésta es el campo escalar de la presión. Para resolverla, se partirá nuevamente de la ecuación de descomposición de Helmholtz-Hodge:

$$u = u_{sol} + \nabla \phi$$

Primero se la simplificará aplicando el operador de divergencia (***div***) en ambos lados. El cálculo de la divergencia de un vector sin divergencia, da como resultado 0. Por lo tanto la ecuación se simplifica de la siguiente manera:

$$\nabla \cdot u = \nabla \cdot (u_{sol} + \nabla \phi)$$
$$\nabla \cdot u = \nabla \cdot (u_{sol}) + \nabla \cdot (\nabla \phi)$$

[18] A la rotación se la suele referir como curl.

$$\nabla \cdot u = 0 + \nabla \cdot (\nabla \phi)$$
$$\nabla \cdot u = \nabla \cdot \nabla \phi$$

Finalmente, otra de las identidades del cálculo vectorial **[21]** menciona que la divergencia del gradiente es equivalente al operador laplaciano de un campo escalar, por lo tanto:

$$\nabla^2 \phi = \nabla \cdot u$$

Esta es la ecuación de Poisson, aplicada a la presión del fluido. Ahora es posible computar la presión del fluido, resolviendo esta última ecuación para el campo escalar ϕ.

Método de proyección de Chorin

Como se mencionó anteriormente, el método de proyección de Chorin consiste en dos pasos:

1) Partiendo del timestep **n** , y con una velocidad $\mathbf{u}^n$ (la velocidad en el timestep **n**) se desea calcular una nueva velocidad para el siguiente timestep **n+1**, es decir $\mathbf{u}^{n+1}$. Primero se computa una velocidad intermedia $\mathbf{n}^*$ utilizando la ecuación de Navier-Stokes pero ignorando el término del gradiente de la presión:

$$u^* = -(u^n \cdot \nabla)u^n + f_{buoy}$$

2) Luego del primer paso, el resultado $\mathbf{u}^*$ será un campo divergente, y se procede a restarle el gradiente de la presión (la operación de proyección) para obtener así la velocidad sin divergencia en el siguiente timestep **n+1**:

$$u^{n+1} = u^* - \nabla p^{n+1}$$

Como se puede observar, será necesario utilizar la ecuación de Poisson (presentada anteriormente) para poder resolver el lado derecho de la ecuación:

$$\nabla^2 p^{n+1} = \nabla \cdot u^*$$

Ecuación 15

Mark Harris **[15]** utiliza una notación más amigable para describir el proceso, en donde define los siguientes operadores:

$\mathbb{S}$ = es equivalente a la solución de la ecuación de Navier-Stokes.

$\mathbb{P}$ = es equivalente a la operación de proyección.

$\mathbb{A}$ = es equivalente a la operación de advección.

$\mathbb{F}$ = es equivalente a la adición de fuerzas.

Todos los operadores reciben como parámetro un campo vectorial y retornan uno nuevo. El operador de proyección tiene la particularidad de retornar un campo vectorial sin divergencia. Utilizando estos operadores, el procedimiento general para la resolución puede ser expresado de la siguiente forma, partiendo de un campo vectorial $\mathbf{u}_0$ sin divergencia:

$$\mathbb{S} = \mathbb{P} \circ \mathbb{F} \circ \mathbb{A}(\mathbf{u}_0)$$

Los operadores mencionados se resuelven de derecha a izquierda. Luego de aplicar las operaciones de advección y adición de fuerzas, se posee un campo vectorial con divergencia, la cual será removida luego de la operación de proyección.
En la siguiente ilustración se pueden observar las transformaciones que sufre el vector de la velocidad luego de que cada operador haya sido aplicado:

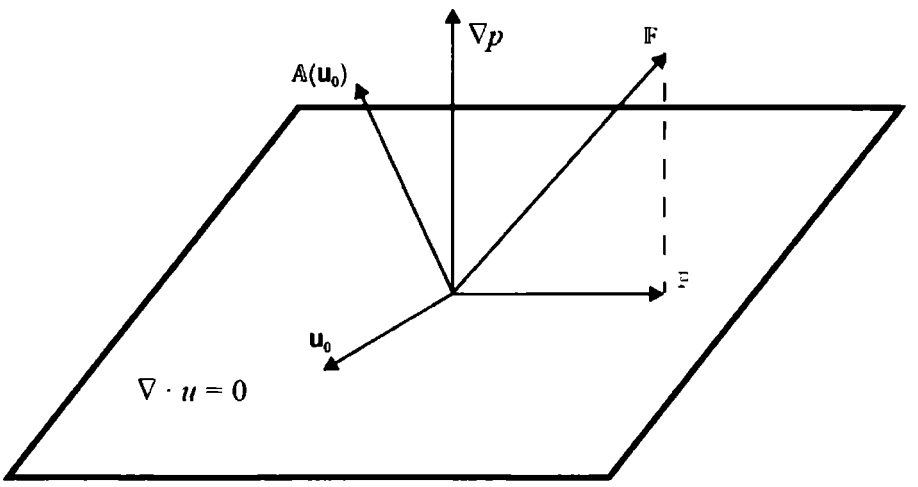


Figura 15: Método de proyección.

En ella, se puede observar que el plano representa el espacio sin divergencia, en el cual se encuentra el vector inicial $\mathbf{u}_0$. Una vez aplicada la advección y las fuerzas externas, el vector se encuentra fuera de la zona libre de divergencia, y por lo tanto se le aplicará la proyección para retornar a ella. Nótese que el resultado de aplicar $\mathbb{P}$ es equivalente al resultado de aplicar $\mathbb{F}$ y luego restarle el gradiente de la presión.

En las siguientes secciones se detallará cómo se realiza la discretización de cada uno de estos operadores, para luego poder pasar a la implementación de las mismas.

7.2.1 Advección

Como se mencionó anteriormente, la advección es el proceso por el cual la velocidad del fluido se transporta a sí misma y a otras substancias inmersas en él (como por ejemplo partículas de humo suspendidas en el aire).

Para la advección se utilizará el método propuesto por Jos Stam en su publicación en 1999, el cual es incondicionalmente estable independientemente del timestep(**Δt**) elegido (a diferencia de publicaciones anteriores como la de Foster y Metaxas **[22]** en donde seleccionar un timestep muy pequeño era necesario para evitar un colapso en la simulación). Gracias a esta estabilidad, es posible no solo usar un *timestep* relativamente grande, sino también uno variable, lo cual lo hace adecuado para simulaciones en tiempo real en donde uno no puede garantizar un timestep estable.

Es importante recalcar que en la advección lo que se busca es calcular un nuevo valor ***q*** (el cual puede representar la densidad, velocidad, temperatura, etc) para una celda en la grilla y avanzar así un paso en la simulación. La operación de advección será representada por la siguiente rutina:

$$\mathbf{q^{n+1} = advect(u, \Delta t, q^n)}$$

Siendo **u** el campo vectorial de la velocidad(discretizado en una grilla), **Δt** el intervalo de tiempo transcurrido entre **n** y **n+1**, y $\mathbf{q^n}$ el valor de **q** en el momento **n**. La rutina retornará el resultado de adveccionar $\mathbf{q^n}$ mediante la velocidad **u** durante el intervalo de tiempo **Δt.**

Para resolver la rutina **advect** se utilizará el método propuesto por Stam conocido como semi-lagrangian advection (también llamado backwards advection), y conceptualmente funciona de la siguiente manera:

Para calcular el valor de **q** en una posición **x** en el momento **n+1**, se retornará el valor de **q** de la partícula(en el momento **n**) que al ser adveccionada quede situada en la posición **x**.

Sin embargo esta descripción asume un punto de vista Lagrangiano (recordar los puntos de vista lagrangiano y euleriano), el cual se lo puede modelar utilizando un sistema de partículas las cuales tienen atributos como la posición, velocidad, densidad, etc; y adveccionando las partículas se avanza en la simulación **[40]**. Sin embargo, es posible adaptar este razonamiento a una grilla, obteniendo así el método semi-lagrangiano.

Si se posee un punto $\mathbf{x_1}$ en la grilla y se desea calcular el valor de $\mathbf{q^{n+1}}$ en la posición $\mathbf{x_1}$, por la definición de advección se sabe que si una (hipotética) partícula con un valor $\mathbf{q_1}$ luego de moverse queda situada en la posición $\mathbf{x_1}$, entonces $\mathbf{q^{n+1} = q_1}$.

Para obtener el valor de $\mathbf{q_1}$ en la posición de la grilla $\mathbf{x_1}$, se tomará la velocidad **u** en la posición $\mathbf{x_1}$, se la invertirá y se retornará el valor de $\mathbf{q^n}$ que haya en la posición $\mathbf{x_2}$.

En otras palabras, si la función **Q(x)** retorna el valor de **q** ubicado en la posición **x** de la grilla, entonces:

$x_2 = x_1 - u$

$q^{n+1} = Q(x_2)$

Sin embargo, continuando con la analogía de la partícula, al estar trabajando sobre una grilla sucederá que la posición resultante x_2 no quede exactamente centrada en la celda (asumiendo el uso de una cell-centered grid) y por lo tanto el valor a retornar de q^n no sería el indicado, ya que en la celda el valor de q^n se almacena en el centro mientras que la posición x_2 no se encuentra en él.

Esto puede ser solucionado realizando una interpolación entre el valor de q^n en la posición de x_2 y los valores de q^n de sus 4 vecinos más cercanos, llamada bilinear interpolation en el caso de trabajar con una grilla de dos dimensiones[19], ilustrado en la siguiente figura:

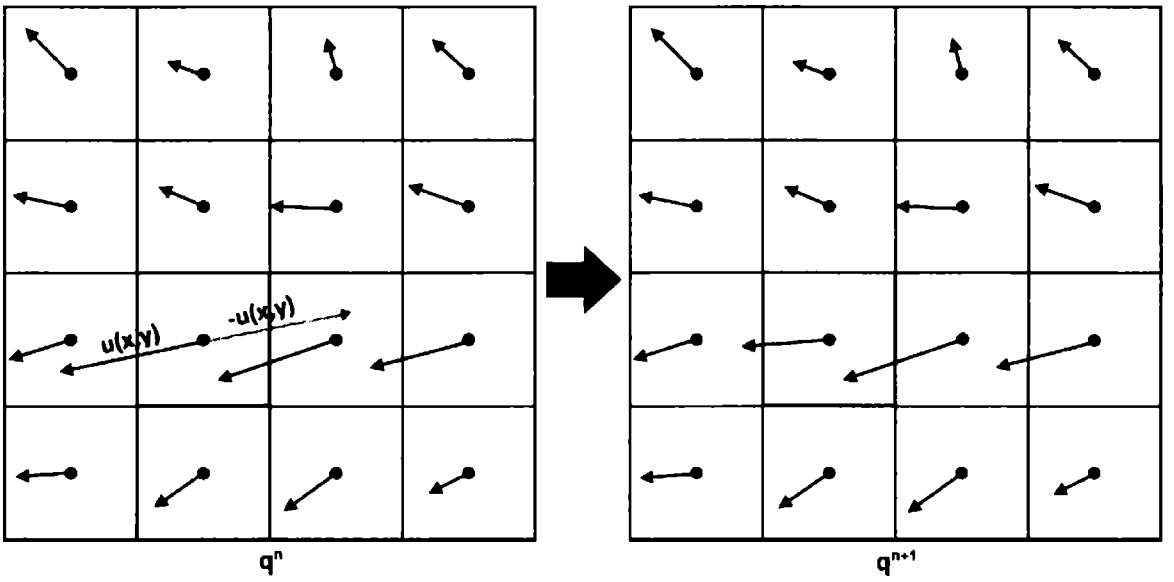


*Figura 16: Actualización de la velocidad **u** de una celda (indicada en rojo).*

No se debe olvidar que la partícula es puramente hipotética y que durante la implementación se utilizará una grilla en lugar de un sistema de partículas.

Dado que se toma un acercamiento Lagrangiano para solucionar el problema de la advección Euleriana, este método es conocido como advección semi-Lagrangiana. La advección semi-Lagrangiana produce mejores resultados cuando el campo vectorial de la velocidad es incompresible. A causa de esto la advección es el primer paso de la simulación, ya que el último paso (la proyección) da como resultado un campo de velocidad libre de divergencia, y por lo tanto incompresible.

Finalmente, en la implementación es necesario realizar la advección de la velocidad, densidad del humo y temperatura. Cada una de estas variables está representada por una grilla, en donde el centro de cada celda contendrá el valor de dichas variables.

[19] Trilinear interpolation en el caso de tres dimensiones.

7.2.2 Adición de fuerzas

Como se mencionó anteriormente, las fuerzas externas están determinadas por la ecuación de la flotabilidad, la cual utiliza la densidad del humo y la temperatura. Dado que la advección de la velocidad debe hacerse sobre un campo libre de divergencia, la adición de fuerzas debe hacerse una vez actualizada la velocidad. Comúnmente se realizan ambas operaciones en el mismo paso, primero la advección semi-lagrangiana y luego la adición de fuerzas. En otras palabras la advección puede constar de dos etapas, por cada celda en la grilla de velocidad:

1) Computar la nueva velocidad $\mathbf{u}_1$ mediante la advección semi-Lagrangiana
2) Agregar las fuerzas de la flotabilidad a la velocidad $\mathbf{u}_1$

En la segunda etapa si se desea también se puede incluir otro tipo de fuerzas externas, como por ejemplo aquellas provocadas por el input del usuario.

Una vez actualizada la velocidad, se procede a realizar la advección de la densidad del humo y la temperatura, utilizando el mismo método de advección.

7.3.3 Proyección de la presión

Para la proyección de la presión primero es necesario resolver la ecuación de Poisson previamente mencionada, y para esto se utiliza un método iterativo llamado Método de Jacobi.

A continuación se presenta la discretización del operador laplaciano, luego se resuelve la ecuación de Poisson, se hace una breve descripción de los métodos iterativos, y finalmente se aplica el método iterativo de Jacobi para construir el campo escalar de la presión.

Discretización del operador laplaciano

El operador laplaciano se discretiza de la siguiente forma utilizando el método de diferencia finita central:

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\Delta x)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\Delta y)^2}$$

En donde Δx y Δy representan la distancia entre cada celda de la grilla.
Si se asume que $\Delta x = \Delta y$ entonces la ecuación se simplifica de la siguiente manera:

$$\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$$

Solución de la ecuación de Poisson

Como se mencionó anteriormente, se sabe que:

$$\nabla^2 p = \nabla \cdot \vec{u}$$

Se utilizará la letra **b** para representar la divergencia de la velocidad en favor de una notación más compacta y amigable. Partiendo de dicha igualdad, se llega a la siguiente ecuación:

$$\frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2} = b$$

Lo que se desea calcular es el punto $p_{i,j}$ por lo tanto se despejará dicho término de la siguiente forma:

$$p_{i,j} = \frac{-b(\Delta x)^2 + p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1}}{4}$$

Ecuación 16

Para computar el campo de la presión se debe aplicar la última ecuación en cada una de las celdas de la grilla, usando el método iterativo de Jacobi.

Métodos iterativos

Los métodos iterativos son utilizados para resolver ecuaciones lineales, partiendo de una aproximación inicial $\mathbf{x}^{(0)}$ y repitiendo el proceso de solución las veces necesarias hasta llegar a una convergencia $\mathbf{x}^{(k)}$ satisfactoria, en donde cada paso da como resultado una solución refinada utilizando el resultado del paso anterior como la nueva aproximación inicial:

$$\mathbf{x}^{(0)} \rightarrow \mathbf{x}^{(1)} \rightarrow \mathbf{x}^{(2)} \rightarrow \ \ldots \rightarrow \mathbf{x}^{(k)}$$

La cantidad de iteraciones a realizar puede ser determinada inicialmente y/o puede ser calculada en base al margen de error en la aproximación, en otras palabras detener el proceso cuando el error está por debajo de un número aceptable (también determinado inicialmente).

Los métodos iterativos son útiles cuando se posee un sistema lineal con un gran número de variables, como lo es en este caso, mientras que los métodos directos serían inadecuados o prohibitivos dada la complejidad computacional que presenta un sistema de ecuaciones con dichas características, ya que intentarían resolver el sistema de ecuaciones con la forma **Ax**

= **b** calculando la inversa de **A** la cual es una operación altamente costosa para un sistema en tiempo real.

La ecuación de Poisson puede ser representada en la forma **Ax = b**, en donde **b** es un vector de constantes que se conocen (en este caso la divergencia en cada una de las celdas), **x** es un vector con las incógnitas a resolver (habrá tantas incógnitas como celdas en la grilla) y **A** es una matriz, que en este caso está implícitamente representada por el operador laplaciano ∇^2.

Al igual que Harris, se utiliza el método iterativo de Jacobi dado que es altamente paralelizable y simple de implementar en la GPU. El método converge de forma relativamente lenta a comparación de otros como el de Gauss-Seidel**[23]** o Multigrid**[2]**, necesitando aproximadamente 50 iteraciones para llegar un error aceptable**[19]**.

En el método de Jacobi, para calcular el valor $\mathbf{p}_{i,j}^{k+1}$ se deben poseer todos los valores de $\mathbf{p}^k$ excepto $\mathbf{p}_{i,j}^{k}$. Esto implica que se deben emplear dos grillas, una contiene los valores del resultado de la iteración **k**, y la otra grilla contiene los nuevos valores de **k+1**. Por lo tanto en la implementación se debe ir intercambiando las grillas, de forma tal que si en una iteración **k** la grilla **A** se utiliza para leer y la grilla **B** se utiliza para escribir, en la iteración **k+1** la grilla **B** será usada para leer, y la grilla **A** para almacenar los nuevos resultados.

En conclusión, para computar el campo de la presión se utiliza la ecuación de Poisson y el método iterativo de Jacobi, configurandolo para realizar 50 iteraciones[20] y con una aproximación inicial de 0, utilizando dos grillas alternadamente entre cada iteración.

7.2.4 Condiciones iniciales y de frontera

Dos temas de suma importancia no han sido mencionados hasta el momento, el estado inicial de la simulación y las condiciones de frontera.

Para poder modelar la evolución del fluido a lo largo del tiempo, por un lado, es necesario contar con el estado inicial de la velocidad, presión, densidad de humo y temperatura, a los cuales en este caso se les asigna el valor cero.

Por otro lado, las condiciones de frontera definen el comportamiento del fluido en los límites del dominio de la simulación. Este tipo de condición es común al momento de resolver ecuaciones diferenciales en un espacio discreto. Dado que el fluido es simulado en una grilla fija en el espacio[21] y que no se produce ningún tipo de interacción con el exterior, se puede decir que la simulación es del tipo "fluido dentro de una caja" y por lo tanto no se produce ningún tipo de intercambio de fuerzas o densidades con el exterior.

---

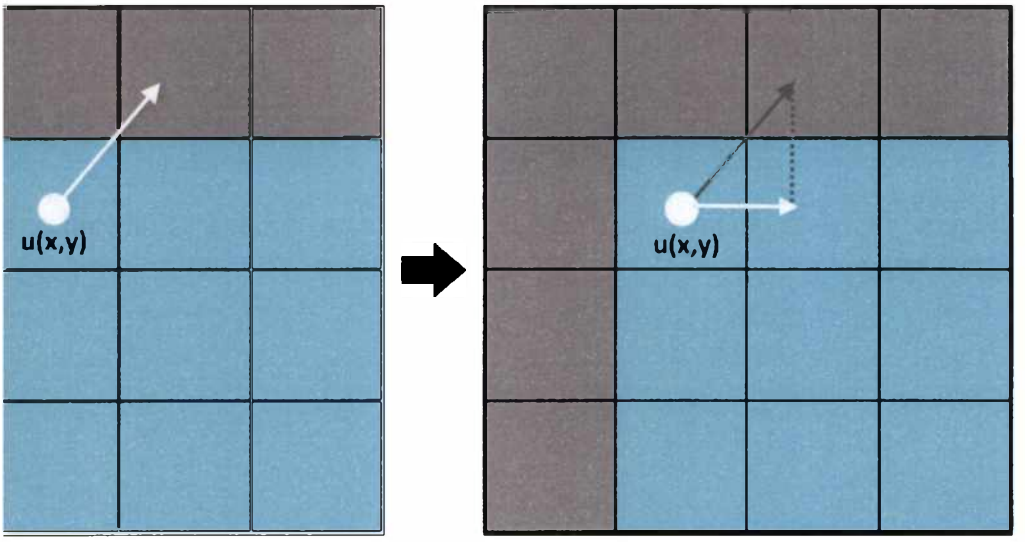
[20] Harris recomienda utilizar entre 40 y 80 iteraciones, y no menos de 20. Por otro lado, Stam demostró que para una aplicación sencilla, 54 iteraciones son suficientes para el contexto en el que será utilizado**[15]**.

[21] En la publicación **[13]** se presenta una grilla la cual puede ser trasladada en el espacio, haciendo que el fluido reaccione al movimiento.

ıtilizando una condición, conocida como free-slip
fronteras la velocidad del fluido con respecto al
ındo así que el fluido escape. Es decir:

ı • n = 0

es el vector normal de la frontera. A este tipo de
richlet, la cual se caracteriza por asignar un valor
]. En otras palabras, utilizando esta condición no
o salga del dominio de la simulación, pero sí se
ma lateral por la frontera, como se ilustra en la



ndo la condición de Dirichlet, evitando así que el fluido escape.

ıacta de forma directa a la manera en la que se
ɪ, dado que es el último paso en la simulación en
ı de realizar el cálculo del gradiente de la presión,
del vector normal de la frontera, dando como
ntacto con las "paredes" de la simulación.

la divergencia y de la presión también se verán
ı lo que sucede con la velocidad y la presión fuera
ɛ frontera que se utiliza indica que no se producirá
, y esto permite asumir que la velocidad fuera del
ría utilizar una velocidad perpendicular al vector
speta con dicha condición. Por lo tanto, durante el
s de velocidad que estén por fuera del dominio,

Por último, durante el cálculo de la presión y de la proyección, dado que no se puede conocer la presión que se encuentra fuera del fluido, se utiliza el valor de la celda central para realizar la diferenciación, permitiendo realizar una aproximación al valor deseado.

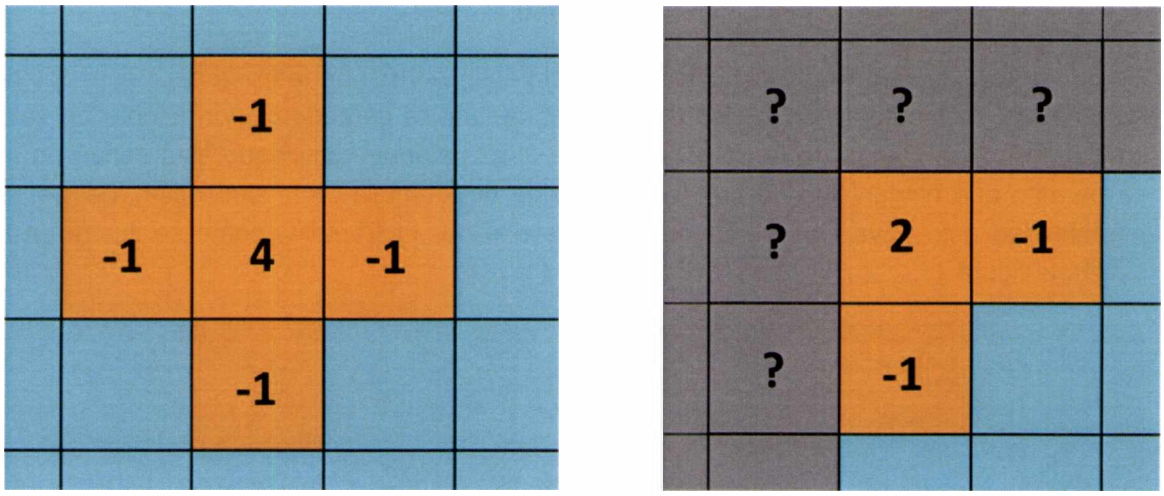


Figura 18: Este tema será discutido nuevamente durante la implementación de las ecuaciones.

8 Simulación de Fluidos: Implementación

En este capítulo se describe la implementación de la herramienta para la simulación de fluidos. La implementación está compuesta por dos partes: la simulación del fluido, y su visualización, conformando así la totalidad de la herramienta a desarrollar.

Para la simulación se ha decidido utilizar la API de DirectCompute para ejecutar programas de propósito general en la GPU (GPGPU). Cabe mencionar que DirectCompute es parte de la familia de APIs de DirectX, creado por Microsoft, y por lo tanto el programa solo puede correr sobre el sistema operativo Windows, con la ventaja de no depender del fabricante de la placa de video.

Para la visualización se utiliza Unity3D, un motor para el desarrollo de aplicaciones interactivas que posee integración con DirectCompute y DirectX.

En las siguientes secciones se introduce el modelo de programación en la GPU, así como también el uso de DirectCompute y la implementación de las ecuaciones desarrolladas en los capítulos previos. Luego se hablará de cómo utilizar los resultados de la simulación y visualizarlos en la aplicación interactiva.

8.1 Simulación

Dado que la GPU juega un papel central en la implementación, a continuación se hará una breve introducción a los conceptos básicos de su arquitectura para poder familiarizarse con el entorno sobre el cual se irá a desarrollar la herramienta. Por motivos de simplicidad se hablará particularmente en el contexto de la arquitectura Fermi **[37]** de las GPUs producidas por NVIDIA (lanzada en abril del 2010). Sin embargo, los conceptos presentados pueden ser encontrados de manera similar en las placas de otros fabricantes.

8.1.1 GPU

Modelo de programación

En el modelo de programación se pueden identificar dos tipos de código dependiendo de en donde sea ejecutado.

Por un lado, se encuentra el código ejecutado de forma serial en un thread de la CPU, referida como host. Por otro lado se encuentra el código que compone el kernel, definido por una función que se ejecutará de forma paralela en múltiples threads de la GPU, referida como device, y que será invocada por el host.

El trabajo del host es el de inicializar y transferir todos los recursos necesarios hacia la memoria del device, los cuales luego serán consumidos por el kernel durante su ejecución. Una vez terminada su tarea, debe transferir los resultados de regreso a la memoria del *host*.

A la hora de construir el kernel, el programador debe decidir la cantidad de threads que irá a utilizar durante su ejecución. Para esto, los threads se encuentran organizados bajo la siguiente jerarquía:

- Threads: la unidad básica de ejecución, en donde se ejecutará una instancia del kernel.
- Blocks: representan un conjunto de threads.
- Grids: representan un conjunto de blocks.

Se debe indicar la cantidad de bloques por grilla, y la cantidad de threads por bloque. La cantidad total de threads a ser utilizados se calcula de la siguiente forma:

totalThreads = blocksPerGrid * threadsPerBlock

Esta cantidad no puede ser modificada una vez haya comenzado la ejecución del kernel.

El kernel tiene a su disposición el **ID** del thread en el que se está ejecutando, y el **ID** del bloque al que pertenece dicho thread . Estas variables pueden ser utilizadas como identificador único del thread para acceder a recursos utilizando dicho identificador como índice. Por ejemplo: si se desea procesar un arreglo de 1024 elementos, se podría crear una grilla con un solo bloque de 1024 threads, y luego cada kernel utilizará el identificador de su thread (el cual posee el rango de 0..1023) para acceder a cada una de las posiciones del arreglo.

Al momento de especificar la organización de bloques y threads, la grilla puede tener hasta dos dimensiones de bloques, y un bloque puede tener hasta tres dimensiones de threads. Es decir, es posible crear una grilla de, por ejemplo, 2x2 bloques, y cada bloque puede poseer 2x2x2 threads. Por lo tanto el identificador del thread estará compuesto por tres componentes: **threadID.x**, **threadID.y**, **threadID.z**. De igual forma el identificador de bloque tendrá dos componentes, **blockID.x**, **blockID.y**.

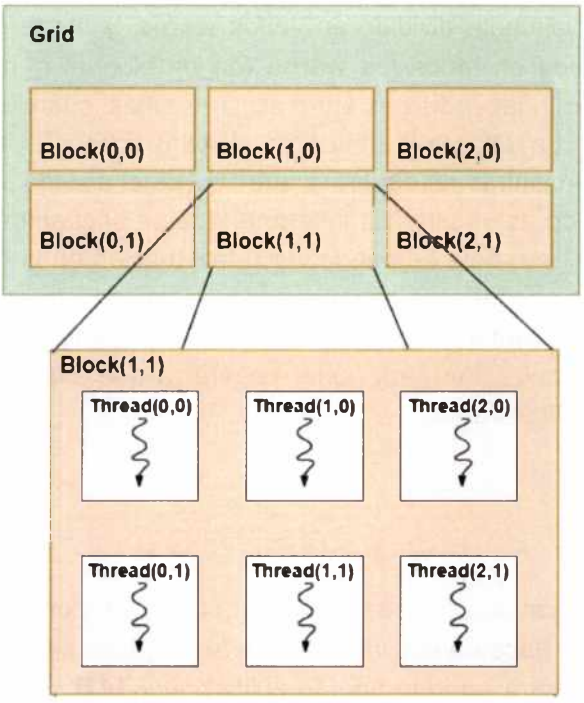


Figura 19: Grid de 3x2 bloques. Cada bloque con 3x2 threads. 36 Threads en total.

Por último, a los threads de un mismo bloque se les provee una memoria compartida y una barrera de sincronización, permitiéndoles comunicarse entre ellos para compartir datos.

Arquitectura

En el lado del hardware se encuentra una jerarquía similar de componentes a los que se mapean los conceptos de thread, bloque, y grid.

Un thread es ejecutado por un Stream Processor (en la actualidad llamado CUDA Core). Un bloque es ejecutado por un Streaming Multiprocessor (SM) el cual está compuesto por varios Stream Processor. Una vez asignado el bloque, éste no será liberado o transferido hasta que finalice la ejecución del kernel. Varios bloques pueden residir concurrentemente dentro de un SM, dependiendo de los requerimientos de recursos de cada bloque.

Cuando un bloque es asignado a un SM, los threads del bloque serán divididos en grupos de 32 threads, llamados **warps**[22]. El **warp** es la unidad de planificación del SM, y todos los threads dentro de un **warp** se ejecutarán de forma paralela bajo el modelo Single Instruction Multiple Data (modelo de la taxonomía de Flynn), es decir que todos los threads ejecutarán la misma instrucción en un momento dado. Es deseable que el programador intente utilizar una cantidad de threads por bloque tal que sea posible completar los **warps** en su totalidad, de lo contrario habrá threads inactivos dentro de un **warp** no podrán ser utilizados dada la naturaleza SIMD de ejecución.

[22] En las arquitecturas provistas por GPUs de AMD, este número es de 64 *threads*, y son llamados *wavefronts*.

Un bloque será potencialmente dividido en varios **warps**, y no se garantiza que la misma instrucción sea ejecutada en todos los **warps** (de un bloque) al mismo tiempo. En otras palabras, los **warps** son independientes entre ellos. Además, cuando un thread dentro de un warp necesita realizar un acceso a memoria, el warp scheduler selecciona otro warp a ejecutarse en su lugar mientras se espera a que finalice el acceso a memoria requerido por el primer warp, ocultando así la latencia inherente que se encuentra al realizar un acceso a memoria. Por lo tanto, fácilmente se puede ver una situación en la cual no todos los warps de un mismo bloque se encontrarán ejecutando las mismas instrucciones. Si los threads de un mismo bloque desean comunicarse entre ellos para compartir algún tipo de dato, deberán usar la barrera de sincronización para poder esperar a que todos los threads estén listos para continuar y compartir los datos.

Jerarquía de memorias

Por último, en esta sección se presentará la jerarquía de memorias que componen la GPU. La simulación de fluidos hace un uso intensivo de la memoria, siendo ésta el principal cuello de botella que se encuentra en este tipo de aplicaciones **[41]**. Si bien no es el objetivo de esta tesina preocuparse por las cuestiones de performance, se debe tener al menos un conocimiento básico sobre la jerarquía de memorias para poder entender a qué se deben los problemas y como pueden ser solucionados.

Se distinguen dos tipos de memorias involucrados en la ejecución de un programa en la GPU: la memoria del host, y la memoria del device. Como se mencionó anteriormente, el host debe realizar la inicialización y la transferencia de recursos desde su memoria hacia la memoria del device, la cual se da mediante el bus PCIe. Dado que las transferencias entre el host y el device son lentas (a comparación de la velocidad de transferencia dentro de la GPU), es recomendable que los datos sean aprovechados tanto como sea posible una vez enviados a la memoria del device. **[25]**

Dentro de la GPU, se encuentran los siguientes tipos de memorias:

- Memoria global (off-chip)
- Memoria de constantes (off-chip)
- Memoria de texturas (off-chip)
- Memoria compartida (on-chip)
- Memoria local (off-chip)
- Registros (on-chip)

A continuación se presentan las características más importantes de cada una de estas memorias.

Registros

Cada SM posee un banco de **registros**, los cuales son distribuidos a los threads ejecutándose dentro de ese SM. En los registros se almacenan las variables locales declaradas en el kernel, y son accedidas únicamente por el thread al cual el registro fue

asignado. En otras palabras, los registros representan el espacio de almacenamiento local de cada thread.

Dado que una instrucción de lectura de estos registros no consume ningún ciclo de reloj, el acceso a este tipo de memoria es el mas rápido de todos. Su tiempo de vida finaliza junto con la ejecución del thread. Es importante recalcar que cada SM posee un número limitado de registros, y existe un número máximo de registros que pueden ser asignados a cada thread. Cuando un thread necesita más registros de los que se le pueden ofrecer, se utilizará la memoria local (de forma transparente al programador) para almacenar los datos restantes.

Memoria Local

La **memoria local** comparte la misma ubicación física (DRAM) junto con la memoria global, memoria de texturas y de constantes. Los datos en la memoria local tienen el mismo alcance y tiempo de vida que el de los registros, pero con la diferencia de que su acceso será mucho más costoso ya que la DRAM se encuentra ubicada off-chip. También es importante mencionar que solo los datos de tipo escalar podrán ser almacenados en los registros, mientras que los arreglos serán almacenados en la memoria local (nuevamente sin intervención del programador). Por ejemplo:

```
float variable; //registro
float variable[10]; //memoria local
```

Memoria Compartida

Los SM además de los registros, también poseen un espacio de **memoria compartida** la cual es distribuida entre los distintos bloques asignados a cada SM. Los threads pueden acceder únicamente a la partición de memoria asignada al bloque al que pertenecen. Una vez que la ejecución del bloque haya finalizado, su memoria es liberada para ser utilizada por otro bloque, por lo tanto el tiempo de vida de la memoria estará ligado al tiempo de vida del bloque. En CUDA, la sintaxis para declarar una variable en memoria compartida es la siguiente:

```
__shared__ float variable; //memoria compartida
```

La memoria compartida ofrece un método rápido y efectivo para permitir que los threads (de un mismo bloque) se comuniquen entre ellos, sin embargo cada SM posee una cantidad limitada de memoria compartida, y se debe tener en cuenta que es posible que más de un bloque esté corriendo al mismo tiempo dentro de un SM. Esto implica que la cantidad de memoria que cada bloque puede utilizar estará dada por la cantidad de bloques asignados en el SM. Por ejemplo si se posee un SM con una memoria compartida de 48KB, y se designaron 6 bloques a correr en dicho SM, cada bloque sólo puede utilizar 8KB (48/6) de memoria. Si un bloque necesita más memoria compartida de la que el SM puede disponer, entonces se verá obligado a utilizar la memoria global.

Si bien la capacidad de almacenamiento es limitada, la memoria compartida provee no solo un costo de acceso similar al de los registros, sino que también un gran ancho de banda dependiendo de la forma en la que se acceda a los datos. Para esto, la memoria compartida está dividida en bancos de memoria de igual tamaño que almacenan una determinada cantidad de palabras de 32 o 64 bits y pueden ser accedidos de forma simultánea. De esta forma, si **N** threads realizan un acceso a **N** bancos de memoria distintos con un mapeo uno a uno, la tasa de transferencia será de **N** veces el ancho de banda de un solo banco. Es decir, si la tasa de transferencia de un banco es de 4 bytes por ciclo de reloj, y se realizan 20 accesos simultáneos a 20 bancos distintos, la tasa de transferencia total sera de 80 bytes en un solo ciclo[23].

Sin embargo, si un banco de memoria es accedido por más de un thread al mismo tiempo, y dichos threads acceden a distintas direcciones dentro del banco , sus accesos son serializados (también conocido como bank conflict) , haciendo que el ancho de banda efectivo se reduzca proporcionalmente a la cantidad de conflictos que se produzcan. Para reducir o evitar los conflictos en el acceso a los bancos, es de suma importancia conocer cómo las palabras se mapean a los bancos de memoria.

Se presentan a continuación cuatro situaciones en las cuales no se produce un conflicto al acceder a los bancos **[45]**:

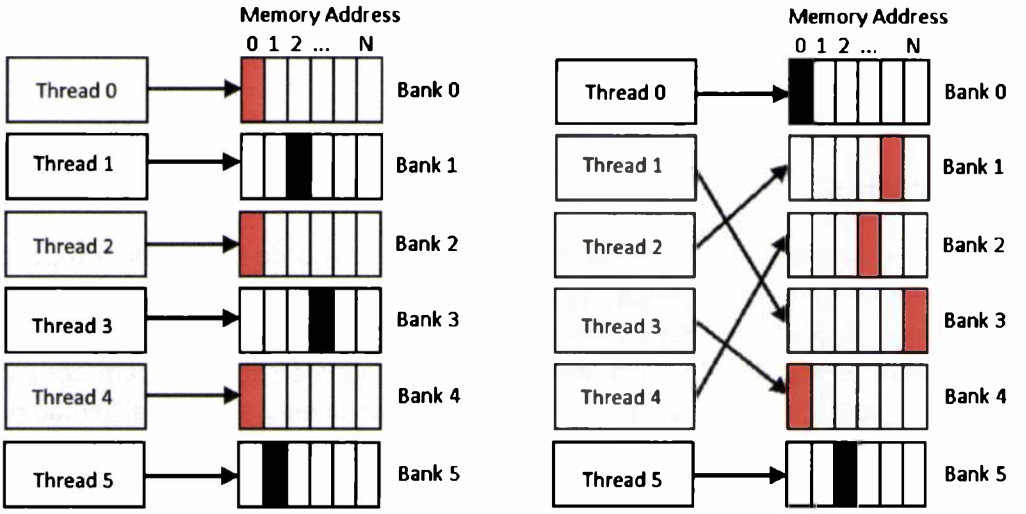


Figura 20: Situación A (izquierda) y B(derecha).

[23] En la arquitectura Fermi, el acceso a una palabra de un banco consume 2 ciclos de reloj.

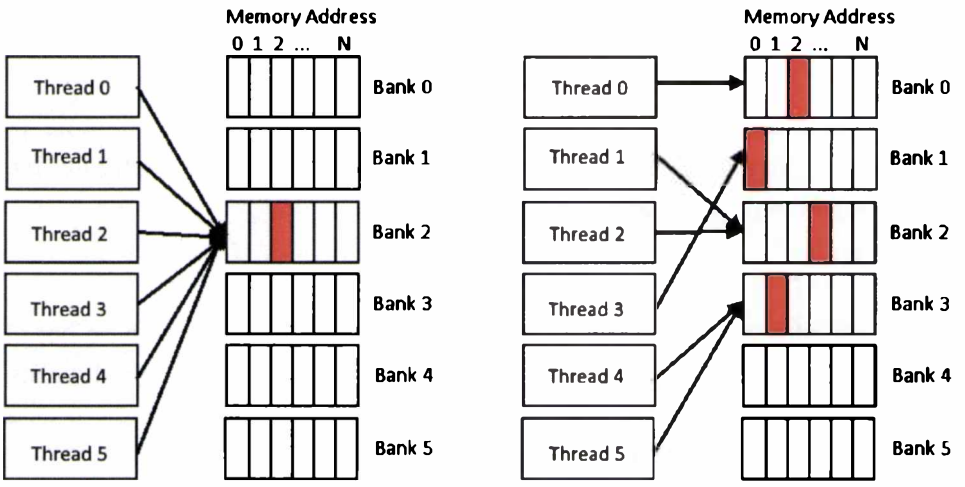


Figura 21: Situación C (izquierda) y D (derecha).

La situación B es un caso especial en el cual todos los threads acceden a la misma palabra en un determinado banco, lo cual resulta en un broadcast del valor. En los casos C y D los bancos son accedidos por más de un thread, pero ambos requieren la misma palabra, por lo tanto no se produce un conflicto. Y el caso A es el más simple en donde nunca habrá probabilidad de conflicto dado que cada thread accede a un banco distinto.

En el caso E, se accede al mismo banco pero a distintas palabras, por lo tanto el acceso será serializado.

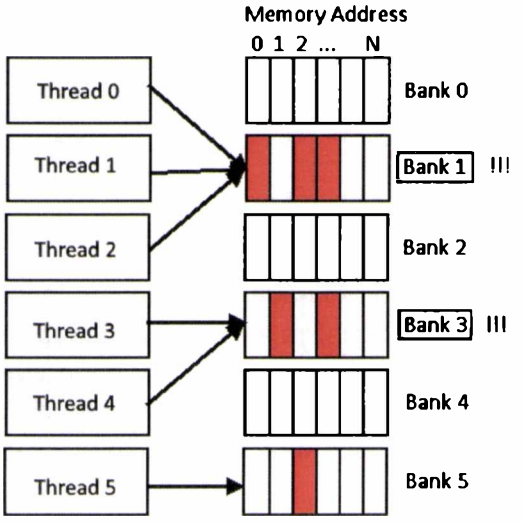


Figura 22: *Situación E, en donde se produce un conflicto en los bancos 1 y 3.*

Por último, es interesante mencionar que la memoria compartida junto con la caché L1 comparten el mismo espacio físico de memoria, y se le da la posibilidad al programador de configurar su particionamiento. Por ejemplo la arquitectura Fermi posee una capacidad total de 64KB a ser distribuida, y se puede seleccionar 16KB o 48KB a utilizar para la memoria compartida. El valor por defecto es de 48KB, y los 16KB restantes serán dedicados a la

cache L1. Esto provee flexibilidad extra a la hora de desarrollar la aplicación, permitiendo favorecer a aquellas aplicaciones que no hagan uso de la memoria compartida pero sí de la caché, o vice versa.

Los datos que necesitan ser cargados en la memoria compartida usualmente provienen como el resultado de alguna operación dentro de los threads, o de la caché L2 la cual a su vez almacena los datos pedidos de la memoria global.

Memoria Global

La memoria global puede ser accedida tanto por los threads del device como por el host. El host, por un lado, la utiliza para enviar los datos que serán consumidos por el device y luego para leer los resultados del programa ejecutado. Por otro lado, los threads la utilizan para adquirir los datos que deberán procesar durante su ejecución, o para comunicarse con threads de distintos bloques. Una vez que una variable es asignada en la memoria global, permanecerá ahí durante toda la vida de la aplicación, hasta que el host decida liberarla. Esto permite que se puedan ejecutar múltiples kernels sin necesidad de volver a transferir los datos en cada ejecución.

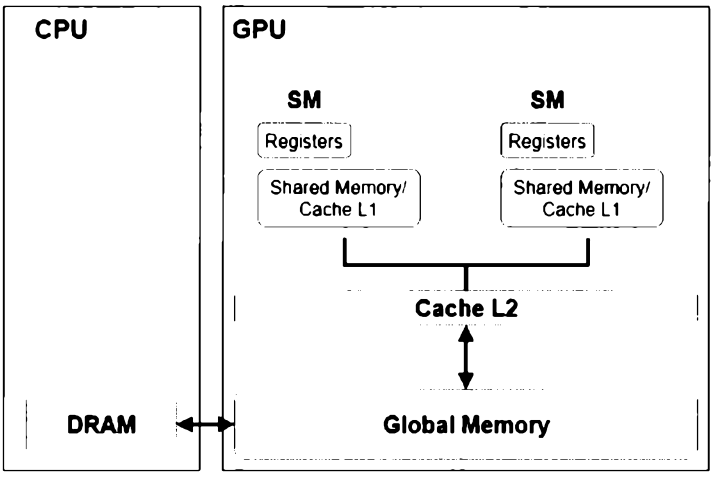


Figura 23: Memoria global.

La memoria global puede ser considerada la memoria "principal", ya que en ella se almacenan todos los datos que el kernel irá a utilizar, y cuando comience su ejecución todos los threads necesitarán acceder a ella. Sin embargo, dado que la memoria se encuentra off-chip, su acceso es extremadamente lento a comparación de las otras memorias[24]. Por lo tanto es crítico que su acceso se haga de forma eficiente, y esto se consigue utilizando el patrón de acceso adecuado.

El patrón de acceso se refiere a la forma en la cual los threads acceden a los datos de la memoria global. Cuando los threads dentro de un warp realizan la petición de una dirección de memoria, todas las direcciones son agrupadas y convertidas en peticiones de líneas de caché. La caché L1 posee líneas de 128 bytes con direcciones alineadas, esto implica que si los 32 threads de un warp realizan la petición de un dato de 4 bytes (el tamaño de un float)

[24] Teniendo una latencia de entre 400 y 800 ciclos de reloj en la arquitectura Fermi.

que se encuentren ubicados de forma contigua en la memoria global, sólo es necesaria una línea de caché para traer el total de 128 bytes. Sin embargo, si los primeros 16 threads acceden a las direcciones de bytes en (por ejemplo) el rango 128..192, y los otros 16 threads acceden a las direcciones 256..320, entonces se traen dos líneas de caché con el contenido de las direcciones desde 128..384, desperdiciando efectivamente media línea de cache en cada una. En otras palabras, se necesitan transferir 256 bytes de los cuales solo 128 son utilizados, a este tipo de acceso se le llama acceso no unificado (non-coalesced access). En el peor de los casos, los 32 threads accederán a direcciones ubicadas en 32 líneas de caché distintas, resultando en una transferencia total de 32*128 bytes.

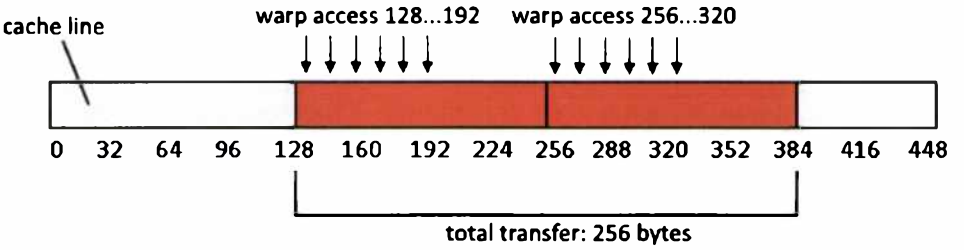


Figura 24: Acceso no alineado a dos líneas de caché.

La caché L2 (ubicada off-chip), a diferencia de la caché L1, utiliza líneas de un tamaño de 32 bytes a las que se les llamará segmentos **[46]**. Cuando los threads de un warp requieren 128 bytes de direcciones contiguas y la caché L1 no los posee, la caché L2 realizará 4 transacciones de segmentos hacia la caché L1 para transferir los 128 bytes.

En CUDA, se ofrece la posibilidad de deshabilitar por completo la caché L1, haciendo que todos los accesos de memoria se dirijan directamente a la caché L2 **[26]**. Esto hace que en el peor de los casos, si 32 threads desean acceder a direcciones ubicadas en líneas de caché distintas, la transferencia total será de solo 32*32 bytes.

Dado que en esta tesina se utiliza DirectCompute en lugar de CUDA, y DirectCompute no provee la posibilidad de seleccionar la forma en la que se realizará la carga de la memoria en caché, no se profundizará más en el tema. Sin embargo se puede apreciar la importancia de realizar accesos a memoria de forma ordenada y a direcciones contiguas para utilizar la menor cantidad de líneas de caché posibles. Esto reduce los accesos a la memoria global, y mejora la performance general de la aplicación. A modo de ejemplo, para ayudar al acceso ordenado y contiguo de memoria, se podría utilizar arreglos de estructuras en lugar de estructuras de arreglos al momento de planificar la organización de los datos en memoria.

Memoria de constantes

La memoria de constantes es utilizada para almacenar valores que no cambiarán durante toda la ejecución del kernel, pero sí entre distintas ejecuciones. Esta memoria solo puede ser escrita por el host antes de que comience la ejecución del kernel, y leída por todos los threads. Se encuentra ubicada en la DRAM, y una vez inicializada la variable permanecerá en la memoria hasta que el host la libere o la aplicación llegue al fin de su ejecución. Dado que la memoria se encuentra off-chip, para acelerar su acceso cada SM posee una caché

de constantes especialmente diseñada para transmitir (broadcast) una dirección de memoria a todos los threads de un warp. Cuando todos los threads acceden a la misma dirección de memoria y ésta se encuentra en la caché, su tiempo de acceso es similar al de los registros. Sin embargo si dicha dirección no se encuentra en la caché, su tiempo de acceso será el mismo que el de la memoria global.

Memoria de texturas

La memoria de texturas fue originalmente diseñada para el pipeline de renderizado de OpenGL y DirectX, pero presenta propiedades que pueden ser útiles para el cómputo de propósito general en la GPU, particularmente en aquellas que trabajan con procesamiento de imágenes. Al igual que la memoria de constantes, la memoria de texturas solo puede ser escrita por el host y leída por los threads (con algunas excepciones). También se encuentra en la DRAM, posee una caché en cada SM, y su ciclo de vida será el de la aplicación o hasta que sea liberada por el host, permitiendo la reutilización de las texturas entre distintas ejecuciones del kernel.

La caché de texturas fue diseñada teniendo en cuenta su uso en aplicaciones gráficas, y es por ésto que fue optimizada con respecto a la localidad espacial en el sistema de coordenadas de la textura, la cual usualmente es de dos dimensiones, pero también podría ser de 1D (como un arreglo) o 3D. Generalmente al trabajar con procesamiento de imágenes, cuando un thread accede a un texel (texture element) es altamente probable que también se accedan a sus elementos vecinos en términos del sistema de coordenadas de la textura, como se muestra en la siguiente ilustración:

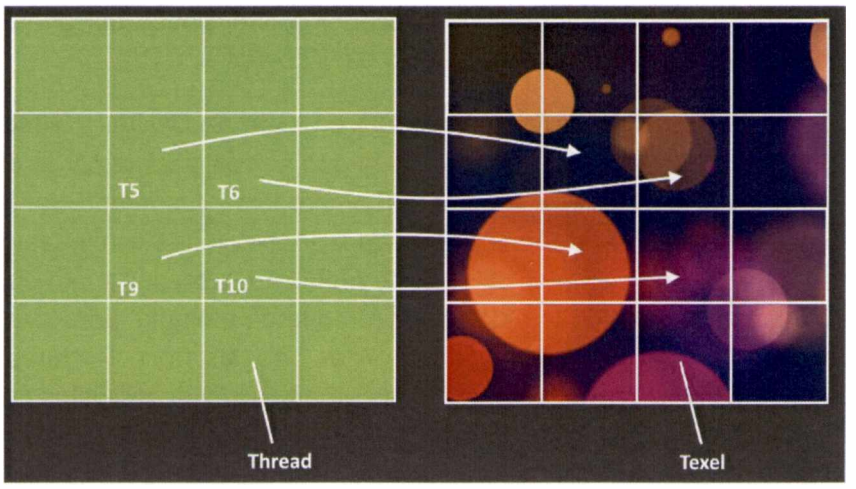


Figura 25: Mapeo de threads a texeles de una imagen.

Una elemento en la textura puede ser referenciada de dos formas: utilizando el índice del elemento del arreglo, o utilizando un conjunto de coordenadas en el rango 0..1 de 1, 2 o 3 dimensiones dependiendo de las dimensiones de la textura. Por ejemplo en el caso de una textura 2D de 3x3 elementos, si se desea acceder al elemento ubicado en el centro sin importar la cantidad de elementos horizontales o verticales que se contengan, se utilizarán las coordenadas <0.5, 0.5> y la caché de texturas se encargará de localizar el elemento que

se corresponda a esas coordenadas, en este caso el elemento ubicado en la posición [1][1] como se muestra en la ilustración 2:

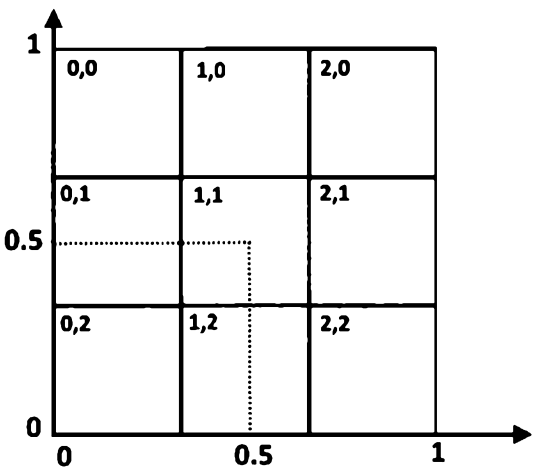


Figura 26: Acceso al elemento (1,1) de la textura utilizando coordenadas en el rango 0..1.

Cuando se utilizan coordenadas en el rango 0..1 para leer un elemento, la caché provee la opción de realizar una interpolación con los valores de los elementos vecinos, por medio de unidades de hardware dedicadas a la interpolación.

La construcción de las texturas no puede ser de forma arbitraria, y solo se pueden utilizar las configuraciones provistas por la API. El tipo de dato de cada uno de los elementos puede ser un número entero sin signo (8 bits), con signo (16 bits), o un número en punto flotante (32 bits). Al leer un elemento de tipo entero, es posible configurar la lectura para que se realice una conversión a punto flotante de 32 bits con rango de 0..1 en el caso de enteros sin signo, y -1..1 en el caso de enteros con signo.

Otra ventaja disponible es el manejo automático de lecturas de la textura cuando se acceden a coordenadas fuera del rango 0..1 **[27]**. Dos de las configuraciones mas utilizadas, son:

• **Repeat**: Si la coordenada de textura está por fuera del rango 0..1, se utiliza la parte fraccional como nueva coordenada. Por ejemplo si la coordenada es 1.3, la nueva coordenada será 0.3. Esto provoca un efecto de repetición en la textura como se muestra en la siguiente ilustración:

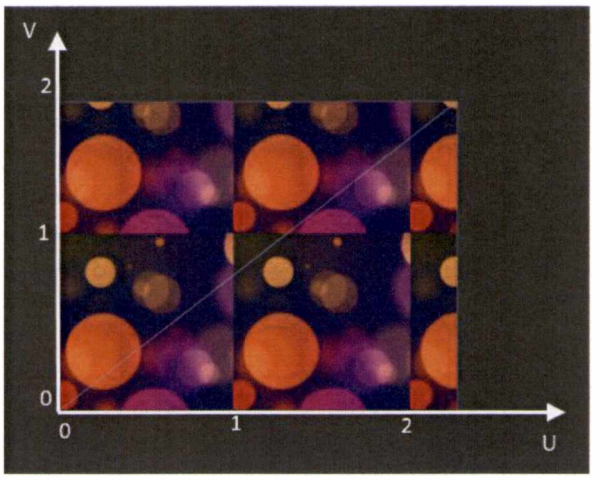


Figura 27: Muestreo de textura configurada para repetir coordenadas fuera de rango.

• **Clamp**: Si la coordenada de textura está por fuera del rango 0..1, la nueva coordenada será el máximo entre la coordenada original, y 0, o el mínimo entre la coordenada original y 1. Por ejemplo si la coordenada a leer es 1.3, la nueva coordenada será 1. Si la coordenada es -1, la nueva coordenada será 0.

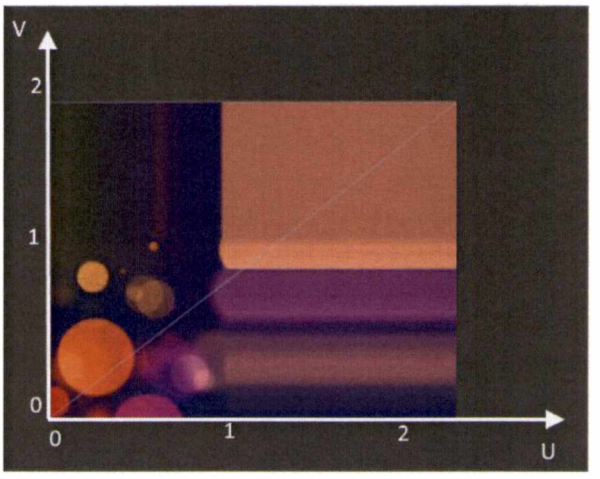


Figura 28: Muestreo de textura configurada para restringir las coordenadas fuera de rango.

Como se puede observar, la memoria de texturas provee funcionalidades particulares orientadas a las aplicaciones gráficas. Sin embargo, para las aplicaciones de propósito general también es posible hacer uso de algunas de estas funcionalidades. Algunos de los casos en los que es ventajoso utilizar texturas en lugar de arreglos convencionales, son:

• El patrón de acceso es difícil de predecir o posee una localidad espacial que se adecúa al acceso de texturas.

• Los datos utilizados en las texturas deben visualizados utilizando el pipeline gráfico.

Anteriormente se mencionó que los threads solo pueden leer de las texturas, sin embargo en el caso de CUDA las memorias 2D también pueden ser escritas por los threads, teniendo

en consideración lo siguiente: la caché de texturas durante la ejecución de un mismo kernel no mantiene una coherencia con respecto a las escrituras de la textura realizadas en la memoria global, y por lo tanto cualquier lectura de una dirección que fue previamente escrita mediante la memoria global durante la misma ejecución del kernel, retornará datos indefinidos. Una vez que finaliza la ejecución del kernel, la caché de texturas es limpiada de forma tal que los threads en la siguiente ejecución de un kernel puedan acceder a los datos actualizados de la textura ubicados en la memoria global. Dicho de otra forma, un thread solo puede leer un elemento en la textura de forma segura si dicho elemento fue actualizado durante la ejecución previa de un kernel, o si fue cargado en memoria por el host **[28]**.

En conclusión, para poder aprovechar al máximo el paralelismo en las GPUs, es fundamental diseñar la aplicación en torno a la arquitectura sobre la cual se ejecuta el código para poder hacer un uso eficiente de los recursos disponibles. De lo contrario, la aplicación podría llegar a desempeñarse de forma marginal con respecto a una implementación en la CPU, o inferior si no se hace un buen uso de los patrones de acceso a memoria, lo cual a su vez implica conocer los tipos de memoria existentes con sus respectivas ventajas y desventajas.

8.1.2 DirectCompute

DirectCompute es parte de la familia de APIs de DirectX, la cual al igual que CUDA permite utilizar los recursos de la GPU para ejecutar programas de propósito general. Es importante mencionar que siguen siendo válidos todos los conceptos y mecanismos presentados anteriormente a cerca de la arquitectura y funcionamiento de la GPU. En el caso de las GPUs de NVIDIA, DirectCompute utiliza la arquitectura de CUDA[25] para su implementación, y si bien no se ofrece exactamente la misma funcionalidad, la mayor parte de ella se conserva. Se debe tener en cuenta que al ser una tecnología propia de Microsoft®, solo puede ser utilizada bajo el sistema operativo Windows®.

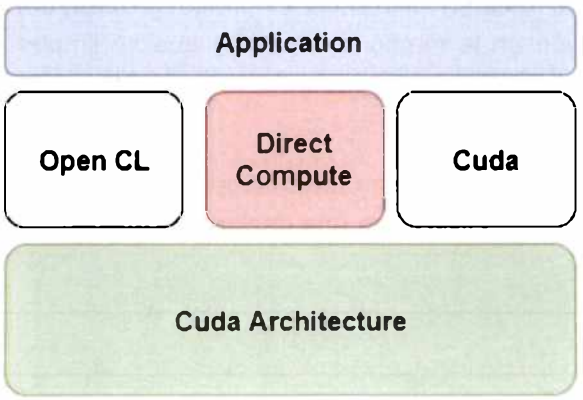


Figura 29: Relación entre DirectCompute y la arquitectura de CUDA.

[25] Utiliza la arquitectura de CUDA en el caso de ejecutarse sobre una placa de video de NVIDIA. En el caso de una placa de video de AMD, se utilizará la arquitectura Graphics Core Next (GCN).

A continuación se mencionan algunas de las ventajas que provee DirectCompute a comparación de otras APIs similares:

• Se encuentra integrado con la API de Direct3D (D3D) la cual es utilizada en el pipeline gráfico. Esto quiere decir que se provee de una inter-operabilidad directa con los recursos gráficos que se encuentran en D3D (texturas, buffers, etc).

• Incluye todas características que se proveen de las texturas en el pipeline gráfico (incluyendo cube maps, mip-mapping, y level of detail), facilitando su manejo.

• Los kernels son escritos en el lenguaje HLSL (High Level Shading Language), el cual también es utilizado para programar los shaders de D3D.

• Provee una sola API independientemente del fabricante de hardware gráfico que se esté utilizando (NVIDIA, AMD, etc), garantizando un cierto grado de consistencia en los resultados independientemente del tipo de hardware.

Una de las características más atractivas para esta tesina es la inter-operabilidad con los recursos del pipeline gráfico, dado que durante la ejecución de la herramienta se deben realizar dos tareas:

1) La simulación, la cual es realizada utilizando DirectCompute
2) La visualización, la cual toma los resultados de la simulación y los visualiza

Por lo tanto, la simulación debe operar sobre recursos que también puedan ser utilizados en la visualización. En DirectX 11 se introdujo, entre otras cosas, un nuevo tipo de recursos que tienen la particularidad de permitir tanto la lectura como escritura por el mismo kernel, entre ellos: **RWTexture1D**, **RWTexture2D**, **RWTexture3D**. En su declaración, se debe especificar el tipo de datos que se almacena en la textura, por ejemplo:

```
RWTexture2D<float3> velocity; //float3 es un vector de 3 elementos
```

Las escrituras en este tipo de recursos serán reflejadas para todos los threads de un mismo grupo. Si se declara el recurso utilizando el prefijo **globallycoherent**, se produce una barrera de sincronización en la memoria haciendo que se limpie la caché de texturas de forma tal que todos los grupos puedan ver la actualización, degradando la performance significativamente en el proceso.

Estos recursos son los que se utilizarán para compartir entre la tarea de simulación y la de visualización. A continuación se muestra una declaración típica de un kernel:

Ejemplo de kernel

```
#pragma kernel ExampleKernel //Esta linea es utilizada por el compilador de Unity
RWTexture2D<float> tex;
```

```
[numthreads(32, 32, 1)]
void ExampleKernel(
    uint3 groupId          : SV_GroupID,
    uint3 groupThreadId    : SV_GroupThreadID,
    uint3 dispatchThreadId : SV_DispatchThreadID,
    uint groupIndex        : SV_GroupIndex)
{
    tex [dispatchThreadId.xy] = 5.0;
}
```

En la línea siguiente a la declaración de la textura a utilizar, se puede observar la cantidad de threads en cada dimensión que se irá a utilizar por bloque, en este caso el equivalente a una matriz de 32x32 threads. Luego la declaración del método main el cual recibe como parámetro todos los identificadores existentes, asociados a una semántica que utiliza el compilador para detectar a qué parámetro corresponde cada identificador. En otras palabras, no es el nombre ni la posición del parámetro lo que hará que se reciba, por ejemplo, el identificador del GroupID, sino la semántica asociada al parámetro.

El programa lo que hace es acceder a la textura utilizando las coordenadas del identificador del thread X e Y, y escribir el valor 5. Nótese que el **SV_DispatchThreadID** es el índice que identifica al thread a nivel de dispatch, por ejemplo si se ejecutan 2 bloques de 32 threads de una sola dimensión cada uno, es decir **[numthreads(32, 1, 1)]**, el **SV_DispatchThreadID** poseerá los valores con un rango de **[0..63,0,0]** y podría ser utilizado para acceder a un arreglo o textura de una sola dimensión con 64 elementos.

Por último, sólo resta demostrar el código del lado del host el cual se encarga de comenzar con la ejecución del programa. Para esta tesina se decidió utilizar la herramienta Unity3D dada la integración que posee con la APIs de DirectCompute y Direct3D, así como también la posibilidad de utilizar el lenguaje C# para la programación de la aplicación. A continuación se muestra un método que ilustra la forma en la que se ejecuta el kernel en la GPU:

Ejecución de kernel en la CPU

```
void LaunchKernel(ComputeShader computeShader, Texture2D texture)
{
    int kernelIndex = computeShader.FindKernel("ExampleKernel");
    computeShader.SetTexture(kernelIndex, "tex", texture);
    computeShader.Dispatch(kernelIndex, 2, 2, 1);
}
```

Unity3D provee de una API amigable para interactuar con cada uno de los recursos, en la cual cada método recibe como primer parámetro el ID del kernel que se ejecutará. Las primeras dos líneas son auto explicativas, y en los parámetros de la tercer línea es en donde se declara la cantidad de bloques que irán a ejecutarse en cada dimensión, en este caso 2 en X y 2 en Y. Teniendo en cuenta que la cantidad de threads por bloque es de 32, se ejecutará un total de 4096 (2*2*32*32) threads en total, con la variable `SV_DispatchThreadID` tomando los valores [0..63, 0..63, 0]. Por lo tanto, la textura que se asignó debe tener al menos una resolución de 64x64 elementos, de lo contrario se escribirá en posiciones fuera de rango.

Finaliza aquí la presentación de la información de DirectCompute necesaria para poder implementar la simulación de fluidos. Para concluir con la sección, es interesante notar que DirectCompute ofrece la posibilidad de desarrollar un gran número de diversas aplicaciones, pero su principal destreza yace en aquellas aplicaciones que se encuentran relacionadas con la visualización de datos. Algunos ejemplos para los cuales la tecnología fue diseñada, son:

- Procesamiento de imágenes y videos para aplicaciones orientadas al consumidor.
- Post-procesamiento de imágenes en videojuegos.
- Inteligencia artificial y físicas en videojuegos.
- Efectos de renderizado avanzados, como ray-tracing e iluminación global.

8.1.3 Implementación

La simulación se realiza en dos dimensiones, por lo tanto para la representación de los datos se utilizan las siguientes texturas:

- Velocidad x2, almacenando vectores de dos elementos (X,Y).
- Divergencia x1, almacenando un valor escalar representando la divergencia.
- Presión x2, almacenando un valor escalar representando la cantidad de presión.
- Densidad del humo x2, almacenando un valor escalar.
- Temperatura x2, almacenando un valor escalar.

Cada textura representa uno de los atributos del fluido que deben ser modelados. Nótese que son necesarias dos texturas por cada atributo (a excepción de la divergencia). Esto se debe a que al realizar la advección de la velocidad, temperatura y densidad es necesario contar con el estado de los atributos en el timestep anterior para poder computar los nuevos valores. El método iterativo de Jacobi utilizado para la presión necesita que los resultados sean transferidos de una textura a otra repetidas veces hasta llegar a una convergencia satisfactoria.

A este intercambio de texturas entre cada ejecución se le llama feedback y es común en los algoritmos iterativos. Cada kernel define una textura de la cual lee los valores(**Texture2D**), y otra en la cual escribe los resultados (**RWTexture2D**). Luego, el host en cada ejecución

intercambiará las texturas asignadas al kernel una vez que éste finalice, haciendo que en la siguiente ejecución la textura que previamente fue asignada para escribir, ahora será la que posea los datos a leer, y la textura que previamente fue utilizada para leer será usada para guardar los nuevos resultados. A continuación se presenta un pseudocódigo del host para ejemplificar el proceso:

```
Texture2D velocity_read, velocity_write;// representan el ID de la textura

advect(velocity_read, velocity_write);
tmp = velocity_write;
velocity_read = velocity_write;
velocity_write = tmp;
```

Ahora que se conocen los recursos a utilizar y la forma de utilizarlos, en la siguiente imagen se ilustran los pasos que compondrán la simulación:

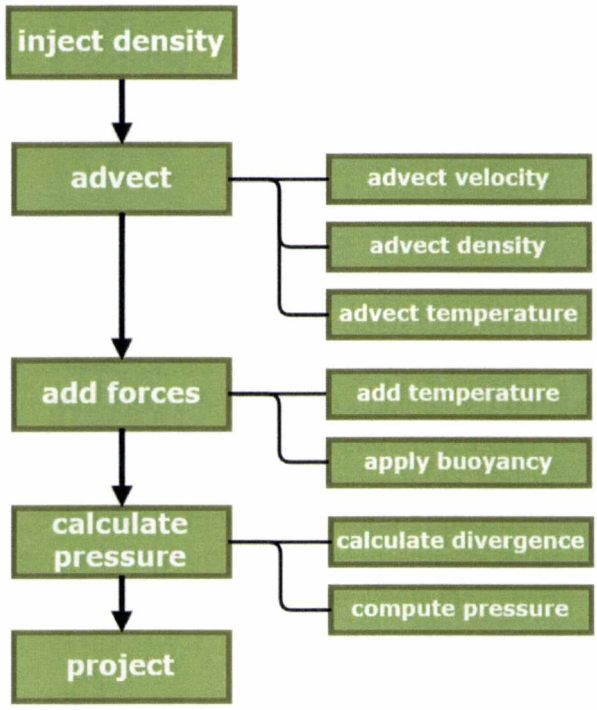


Figura 30: Etapas y sub etapas que componen la simulación.

La primer etapa llamada "inject density" no forma parte de la simulación original, sin embargo es necesaria para establecer la cantidad de densidad inicial de humo que hay en el sistema, así como también para representar cualquier fuente de densidad que pueda existir. Por ejemplo en el caso de un cigarrillo se tendrá una fuente de humo constante en su punta, y por lo tanto se deberá inyectar una nueva cantidad de densidad en cada ciclo de la simulación.

Para cada una de las 5 etapas se utiliza al menos un kernel. En el caso de la advección es posible reutilizar el mismo kernel para adveccionar la velocidad, densidad, y temperatura. En el caso del cálculo de la presión, se debe utilizar un kernel para el cálculo de la divergencia y otro kernel para el cómputo de la presión utilizando el método iterativo de Jacobi, así como también otro para la proyección. Por último la adición de fuerzas puede ser agrupada en un mismo kernel junto con la adición de densidad, lo cual implica que la adición de densidad se realizará después de la advección, pero no será un problema.

En conclusión, se deberán implementar los siguientes kernels:

- Advección
- Adición de fuerzas
- Adición de densidad y temperatura
- Cálculo de divergencia
- Cálculo de presión
- Proyección

El trabajo del host es únicamente el de coordinar la ejecución de cada uno y asignar los recursos correspondientes entre cada ejecución. Una vez terminada la simulación, se utiliza la textura de densidad para ser visualizada en pantalla mediante el pipeline gráfico.

Advección

La implementación de la advección sigue el algoritmo de backwards advection presentado anteriormente, pero con algunas modificaciones. Es necesario que la magnitud de la velocidad se corresponda con el tamaño de la grilla, y para esto habrá que transformar el **`SV_DispatchThreadID`** a valores normalizados en el rango 0..1, de forma tal que si se trabaja en una textura con una resolución de 32x32, el thread con id **[31,31]** obtendrá las coordenadas **(1, 1)**, el thread con id **[16,16]** las coordenadas **(0.5, 0.5)**, etc. Esto a su vez permitirá utilizar las funciones de interpolación de las texturas que se ofrecen como parte del pipeline gráfico. Esta funcionalidad es encapsulada en el método llamado **`normalize_pos(id)`**.

Una vez normalizadas las posiciones de los threads en la grilla, la implementación de la advección no presenta ningún problema. A continuación se muestra el código del kernel, dejando de lado la declaración de los recursos para simplificar su lectura:

Kernel de advección

```
float2 velocity = velocityR[id.xy].xy;   //id : SV_DispatchThreadID

float2 backwards_pos = normalize_pos(id.xy) - velocity * dt;
float2 new_value = source.Sample(LinearClamp, backwards_pos).xy;
target[id.xy] = new_value;
```

La variable **velocityR** es la textura que almacena la velocidad necesaria en la advección. Las variables **source** y **target** también son texturas, que representan el atributo que irá a ser adveccionado, utilizando el **source** para la lectura del valor actual y el **target** para almacenar los resultados. Si bién las variables **velocityR** y **source** son las mismas para la advección de la velocidad, la separación permite que el kernel pueda ser reutilizado para adveccionar cualquier tipo de atributo. Por ejemplo para la advección de la densidad, las variables **source** y **target** contendrán las texturas de lectura y escritura de la densidad.

La variable **dt** (delta time) representa el tiempo transcurrido entre un ciclo de la simulación y el siguiente, o dicho de otra forma, el timestep con el que avanzará la simulación. Por último la variable **LinearClamp** es un objeto que representa la configuración con la que se realizará la lectura en la textura. En este caso, se realiza una interpolación lineal (**Linear**) entre los 4 vecinos mas cercanos a la posición requerida, y en el caso de que uno de los vecinos se encuentre fuera de rango, se utiliza el valor del elemento mas cercano a los límites (**Clamp**). Es importante distinguir entre los dos tipos de acceso a la textura, el primero utiliza un número entero como índice para acceder al elemento, mientras que el segundo utiliza el método **source.Sample** recibiendo como parámetro la configuración del muestreo (**LinearClamp**) y coordenadas en el rango 0..1, retornando el resultado de interpolar entre los 4 vecinos más cercanos a las coordenadas indicadas.

Con esta información, la comprensión del código resulta trivial. Primero se calcula la posición en donde la "partícula" se encontraba en el timestep anterior, utilizando la velocidad inversa de la posición en la que ahora se encuentra. Luego se realiza una interpolación entre los cuatro valores vecinos a la posición calculada, y se almacena el resultado en la textura **target**.

Adición de densidad y temperatura

La adición de densidad y temperatura es controlada por la persona encargada de integrar la simulación con el entorno en el cual se trabaja, permitiéndole decidir la mejor forma de adaptar el fluido en base a las necesidades de la aplicación. Por ejemplo, si en la aplicación se desea modelar humo saliendo de la punta de un cigarrillo, se deberá ubicar una fuente de calor y de densidad en las posiciones que se correspondan con la punta del cigarrillo. Si el cigarrillo está en movimiento, la fuente de calor y densidad deberá moverse junto con él para reflejar el hecho.

Kernel de adición de densidad y temperatura

```
float2 pos = normalize_pos(id.xy);
float amount = 1 - distance(pos,float2(0.5, 0.5))/0.1;
float current_amount = textureRW[id.xy].x;
textureRW[id.xy] = float(current_amount  + amount * dt);
```

Dado que el kernel es utilizado para la adición de densidad y temperatura, el host asigna a la variable **textureRW** las texturas de densidad y temperatura en cada ejecución. El resultado de ejecutar este kernel es un punto de calor y densidad en el centro de la textura que aumentan de intensidad a medida que transcurre la simulación.

Adición de fuerzas

Para la adición de fuerzas se utilizan las texturas de temperatura (**temperatureR**), velocidad (**velocityRW**), y densidad (**densityR**). Luego en el kernel simplemente se computan las fuerzas de flotabilidad presentadas anteriormente. Es importante recordar que al finalizar la ejecución de este kernel, la velocidad resultante poseerá divergencia, y por lo tanto la advección deberá realizarse antes de este paso.

Si se desea, se podrían agregar otros tipos de fuerzas artificiales según se lo necesite, como por ejemplo utilizar la posición y dirección en la que el jugador mueve el mouse, agregando fuerzas que impulsen el fluido siguiendo la trayectoria del mouse.

Kernel de adición de fuerzas

```
float2 up = float2(0,1);
float2 current_velocity = velocityRW[id.xy].xy;
float current_tmp = temperatureR[id.xy].x;
float temperature_term = (current_tmp - ambient_temp) * beta;
float density_term = -alpha * densityR[id.xy].x;
float2 buoyancy = up * (density_term  + temperature_term);
velocityRW[id.xy] = current_velocity + buoyancy * dt;
```

Divergencia

En el kernel de la divergencia se realizan dos tareas: el cálculo de la divergencia propiamente dicho, y la inicialización de los valores de la presión que serán utilizados en la siguiente etapa. Para esto se utilizan las texturas de divergencia (**divergenceW**), presión (**pressureW**) y velocidad (**velocityR**) siguiendo la ecuación 11 (cálculo de divergencia), teniendo en cuenta las condiciones de frontera. Nótese que en aquellas posiciones que se encuentren en los bordes de la grilla, se intentará leer el valor de la celda vecina ubicada fuera de rango, dando como resultado en un valor indefinido. La función **out_of_bounds** recibirá una posición como parámetro y retornará verdadero en los casos en donde la posición se encuentra fuera de la grilla. Como se mencionó anteriormente, se asume que la velocidad fuera de la grilla es de 0, por lo tanto se reemplazará el valor de la velocidad por 0 en aquellas posiciones fuera de rango.

Kernel de divergencia y inicialización de presión

```
float up       = velocityR[id.xy + uint2( 0, 1    )].y;
float down     = velocityR[id.xy - uint2( 0, 1    )].y;
float left     = velocityR[id.xy - uint2( 1, 0    )].x;
float right    = velocityR[id.xy + uint2( 1, 0    )].x;

if(out_of_bounds(id.xy + uint2 (1,0)))
    right = 0;
if(out_of_bounds(id.xy - uint2 (1,0)))
    left  = 0;
if(out_of_bounds(id.xy + uint2 (0,1)))
    up    = 0;
if(out_of_bounds(id.xy - uint2 (0,1)))
    down  = 0;

divergenceW[id.xy] = ((right - left) + (up - down)) /2;
pressureW  [id.xy] = 0;
```

Presión

Para la presión se utilizan las texturas de divergencia(**divergenceR**) y presión (**pressureR**) que se inicializan durante el cálculo de la divergencia. Como se mencionó anteriormente, la presión se calcula mediante la ecuación 16 utilizando el método iterativo de Jacobi, lo cual implica realizar múltiples ejecuciones del kernel, en este caso 50.

Al igual que en la divergencia, se debe tener en consideración el valor de aquellas celdas que estén fuera del dominio de la simulación. Dado que no se conoce el valor de la presión fuera de los límites del fluido, se utiliza el valor de la celda central para poder realizar la diferenciación en aquellas celdas que estén en contacto con los límites del fluido.

Kernel del cálculo de la presión

```
float up       = pressureR[id.xy + uint2( 0, 1    )].x;
float down     = pressureR[id.xy - uint2( 0, 1    )].x;
float left     = pressureR[id.xy - uint2( 1, 0    )].x;
float right    = pressureR[id.xy + uint2( 1, 0    )].x;
float center   = pressureR[id.xy].x;

if(is_boundary(id.xy + uint2(1,0)))
    right   = center;
if(is_boundary(id.xy - uint2(1,0)))
    left    = center;
if(is_boundary(id.xy + uint2(0,1)))
    up      = center;
if(is_boundary(id.xy - uint2(0,1)))
    down    = center;
```

```
float divergence = divergenceR[id.xy].x;
float pressure = (up + down + left + right - divergence) /4;
pressureW[id.xy] = pressure;
```

Proyección

Por último, se implementará el kernel de proyección como se ilustra en la Figura 15, utilizando la textura de la presión (**pressureR**) generada en el paso anterior, para luego calcular el gradiente y restárselo a la velocidad actual (**velocityRW**).

Al igual que en el kernel de la presión, se utiliza la celda central en lugar de aquellas celdas vecinas que estén fuera del dominio de la simulación. Sin embargo se debe recordar que la velocidad con respecto al vector normal de la frontera debe ser de 0. Para poder aplicar correctamente esta restricción, se hace uso de una máscara que indique cuál de los lados está en contacto con la frontera de la simulación, para luego nulificar la velocidad en esa dirección.

Dicho de otra forma, la máscara posee el valor 0 en el componente (**x** o **y**) del vector en cuya dirección donde se encuentre ubicada una celda fuera del dominio, y un 1 si la celda se encuentra dentro del dominio. Luego la velocidad resultante de la proyección es multiplicada por la máscara, y si la celda procesada no se encuentra en contacto con ningún borde entonces la velocidad no sufrirá ningún cambio. Si la celda procesada se encuentra (por ejemplo) en contacto con un borde a su derecha, entonces el componente **x** de la máscara poseerá el valor 0, haciendo que la velocidad en dicho componente sea de 0 al ser multiplicada por la máscara.

Kernel de proyección

```
float up       = pressureR[id.xy + uint2 ( 0,1    )].x;
float down     = pressureR[id.xy - uint2 ( 0,1    )].x;
float left     = pressureR[id.xy - uint2 ( 1,0    )].x;
float right    = pressureR[id.xy + uint2 ( 1,0    )].x;
float center   = pressureR[id.xy].x;

float2 mask = float2(1,1);

if(is_boundary(id.xy + uint2 (1,0))){ right = center; mask.x = 0;}
if(is_boundary(id.xy - uint2 (1,0))){ left  = center; mask.x = 0;}
if(is_boundary(id.xy + uint2 (0,1))){ up    = center; mask.y = 0;}
if(is_boundary(id.xy - uint2 (0,1))){ down  = center; mask.y = 0;}

float2 velocity = velocityRW[id.xy].xy;

float2 new_vel = velocity - float2(right - left, up - down) /2;
velocityRW[id.xy] = new_vel * mask;
```

Concluye así la sección de implementación. El host se encarga de realizar la inicialización de los recursos y de ejecutar los kernels en el orden apropiado. Una vez finalizada la ejecución se utiliza la textura de la densidad del humo para visualizarla en pantalla. A continuación se muestra el código de la función **Update**, la cual es invocada una vez por frame, y en donde se pueden ver las llamadas principales a los métodos que realizan cada una de las etapas de la simulación.

Método Update del Host

```
void Update()
{
    Advect(density_tex_0, density_tex_1);
    Advect(temperature_tex_0, temperature_tex_1);
    Advect(velocity_tex_0, velocity_tex_1);
    AddForces();
    CalculateDivergence();
    SolvePressure();
    Project();
    Visualize(density_tex_0);
}
```

Nótese que se utiliza la misma rutina de advección para adveccionar la textura de densidad, temperatura y velocidad. Dentro de cada rutina luego de finalizar la ejecución del kernel se intercambiarán las texturas (feedback), de forma tal que todas aquellas que contengan el nombre **_0** serán las que siempre posean los resultados de la última operación realizada, y es por ésto que la textura **density_tex_0** será la utilizada para visualizarse en pantalla.

8.1.4 Consideraciones de performance

Una de las mayores preocupaciones al momento de implementar una simulación de fluidos en la GPU es el consumo ancho de banda y, en menor medida en dos dimensiones, el consumo de memoria.

En el caso del consumo de memoria, son las simulaciones en tres dimensiones las que realmente se ven afectadas. Si se desea que la simulación cubra un gran espacio en la escena mientras se mantiene la calidad, se deberá aumentar significativamente la resolución de las texturas para satisfacer este requisito. En dos dimensiones y utilizando una resolución de 512x512 para cada textura (velocidad, densidad, temperatura, divergencia y presión) con una precisión de 32 bits, se calcula un consumo total de 11 MB, mientras que para tres dimensiones y utilizando una textura de 512x512x512 el consumo será de aproximadamente 5.6GB. En la Figura 31 se muestra una tabla que presenta como crece el consumo de memoria a medida que se incrementa la resolución en una simulación de tres dimensiones.

Alex Dunn **[29]** presenta una técnica llamada Sparse simulation, la cual permite desacoplar la complejidad computacional (en términos de consumo de memoria) de la resolución utilizada en las texturas, permitiendo una simulación detallada que abarca grandes espacios. Dado que no fue el objetivo de esta tesina implementar una simulación en tres dimensiones, La discusión de esta técnica queda fuera del alcance.

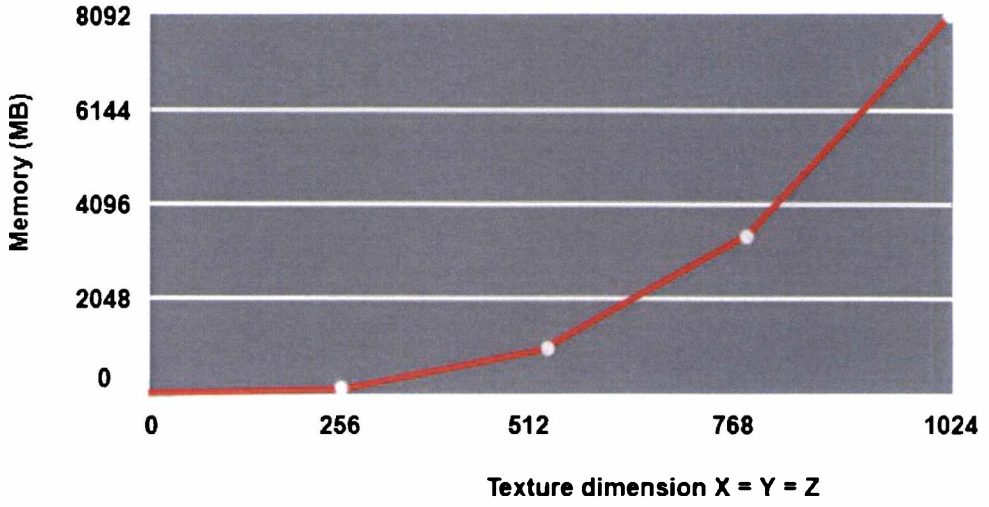


Figura 31: Consumo de memoria para diferentes resoluciones en una textura 3D de 4 canales (RGBA) en punto flotante con 16 bits de precisión.

En lo que respecta al consumo de ancho banda, si bien los kernels son simples desde el punto de vista aritmético y los patrones de acceso a memoria presentan una buena localidad espacial, para poder realizar los cálculos se necesita leer y escribir en memoria múltiples veces generando una alta carga en la transferencia de datos.

Para combatir este problema una de las cosas que se puede hacer es un cambio de precisión de los valores en punto flotante que se almacenan en las texturas, pasando de 32bits a 16bits sin que se observen diferencias notables en los resultados, reduciendo así la cantidad de datos a transferir en un 50%.

Otra optimización que se puede realizar es la de almacenar varios de los atributos del fluido en una misma textura, por ejemplo, utilizando una textura de tipo **float4**, almacenando la velocidad(**float2**), densidad(**float**) y temperatura(**float**). Sin embargo si no se es cuidadoso en la selección de atributos a almacenar en una misma textura, se podría llegar a aumentar la transferencia de memoria. Siguiendo con el ejemplo anterior, en el kernel de adición de densidad se deberán transferir los 4 elementos para poder actualizar la densidad, haciendo que los elementos restantes se transfieran de forma innecesaria. Por lo tanto, es deseable que aquellos atributos que convivan en la misma textura siempre sean utilizados, como por ejemplo la divergencia y la presión, los cuales ambos son accedidos durante el kernel de la divergencia y la presión.

Por último, es posible optimizar significativamente la cantidad de accesos a memoria haciendo uso de la memoria compartida. Primero se declara un arreglo de elementos que será almacenado en la memoria compartida. La longitud del arreglo será equivalente a la cantidad de threads por bloque. Luego, cada thread leerá de memoria su propia posición en

la textura para almacenarla en el arreglo compartido. Una vez hecho esto, se genera un punto de sincronización (utilizando el método **GroupMemoryBarrierWithGroupSync**) permitiendo esperar a que todos los threads hayan almacenado su valor en el arreglo. Por último, cada thread luego de la barrera procederá a acceder a las celdas vecinas según lo necesite, con la diferencia de que este acceso se producirá a los elementos del arreglo compartido y no a la memoria global.

El uso de la memoria compartida es particularmente eficiente en los kernels donde se computen derivadas, como por ejemplo divergencia, presión y proyección, debido a la localidad espacial inherente utilizada en cada tarea. La advección, sin embargo, dado su patrón de acceso "aleatorio" no podrá hacer un buen uso de la memoria compartida, ya que puede darse el caso (durante altas velocidades) en el cual la posición requerida no se encuentre en dicha memoria.

8.2 Visualización

En esta sección se presentan todas las etapas por las que se debe pasar para poder visualizar la textura de densidad en un espacio de tres dimensiones (siendo la simulación en dos dimensiones). Se hace una introducción al rendering pipeline de forma simplificada, enfocándose en el proceso de rasterización. Luego se muestra el código a ser ejecutado por la GPU para poder visualizar los resultados de la simulación.

Se debe mencionar que una gran parte del trabajo de inicialización en la GPU es llevado a cabo de forma interna por Unity3D **[30]**, ofreciendo una API amigable que puede ser utilizada para mostrar rápidamente los resultados, por lo tanto no se detallarán las llamadas de la API de Direct3D que se deban realizar para preparar el renderizado de un objeto (inicialización de buffers, constantes, binding de recursos, programas, etc).

8.2.1 Introducción al Rendering Pipeline

El rendering pipeline (también llamado graphics pipeline) de Direct3D[26] es el mecanismo por el cual los recursos son procesados en la GPU para renderizar una imagen. El pipeline está compuesto por varias etapas, las cuales realizan varios tipos de transformación sobre los datos a medida que éstos progresan por cada una de las etapas, una etapa a la vez. Es necesario entender cómo operan, y sobre qué tipo de datos operan cada una de las etapas para poder implementar los algoritmos necesarios involucrados en la renderización de una imagen y, en este caso, la visualización de los resultados de la simulación. El artículo **[38]** publicado por NVIDIA contiene información acerca de cómo se mapea la arquitectura de hardware de la GPU (mencionada en la sección de simulación) a las etapas lógicas presentadas a continuación.

Los datos son enviados como entrada (input) al inicio del pipeline, en donde son procesados por la primer etapa. Estos datos consisten en variables basadas en vectores de hasta cuatro componentes. Cuando los datos terminan de ser procesados por la primera etapa, son trasladados a la siguiente, permitiendo que la primera etapa quede libre para recibir una nueva porción de los datos de entrada. Esto quiere decir que las primeras dos etapas del pipeline pueden estar trabajando al mismo tiempo de forma paralela. Cuando la segunda etapa finaliza su operación, los datos son pasados a la siguiente, y el proceso se repetirá hasta que todas las etapas del pipeline esten ocupadas. Una vez que todos los datos llegan al final del pipeline, los resultados serán guardados en un recurso de salida (*output*), típicamente una textura, para que luego ésta sea utilizada por el host según lo requiera. Por ejemplo, la textura resultante podría ser utilizada para mostrarla en pantalla, o podría ser consumida como recurso por alguna etapa en una nueva ejecución del pipeline.

[26] Si bien aquí se tratará particularmente el *pipeline* de Direct3D, el de OpenGL es muy similar.

En la Figura 32 se muestran las etapas básicas que componen al pipeline. Direct3D 11[27] posee muchas más etapas de las que se muestran en la figura, además de una mayor flexibilidad en lo que respecta al acceso de recursos **[31]**. Sin embargo, se decidió reducir la cantidad de etapas para enfocarse sólo en aquellas que serán utilizadas en esta tesina.

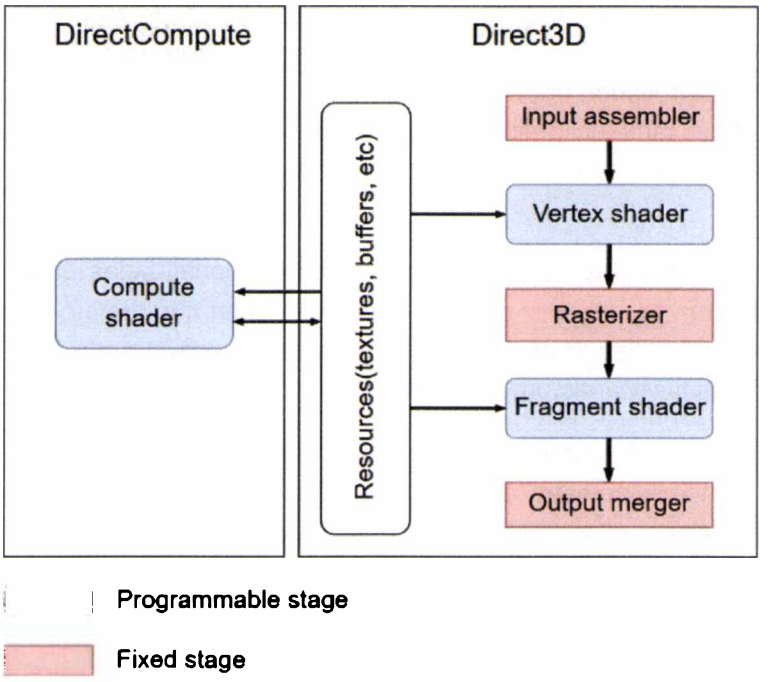


Figura 32: Etapas del rendering pipeline (simplificado).

Cada una de las etapas en el pipeline está destinada a cumplir con un propósito particular, y esta especialización permite que cada etapa pueda ser optimizada de forma mas eficiente que una etapa que realice múltiples tareas.

El trabajo del desarrollador es el de configurar apropiadamente el estado de cada una de las etapas para obtener los resultados deseados. Existen dos tipos de etapas a configurar, etapas fijas (Fixed stage) y etapas programables (Programmable stage, o también llamado Programmable shader stage).

Fixed stage

Las etapas fijas realizan un conjunto fijo de operaciones específicas sobre los datos que reciben. Pese a que pueden ser configuradas de varias formas, la función para la que fueron diseñadas siempre será la misma. Las etapas fijas pueden ser vistas como una función en algún lenguaje de programación, que reciben como parámetro los datos de entrada y las configuraciones posibles que alterarán el comportamiento de la función

[27] En esta tesina se utiliza Direct3D 11 debido particularmente a que Direct3D 12 no posee cambios significativos en su estructura, sino que más bien ofrece un manejo de la API con una granularidad más fina, lo cual hace que aumente la complejidad para ser usada pero con la ventaja de obtener mejoras en términos de performance.

transformando los datos de una forma u otra dependiendo del código definido en el cuerpo de la función. No será posible cambiar el código, pero mediante la configuración será posible cambiar la transformación que sufren los datos.

Programmable stage

En el caso de las etapas programables, como su nombre indica, se puede programar el comportamiento de la etapa mediante pequeños programas escritos en el lenguaje HLSL **[48]** (el mismo utilizado en DirectCompute). A estos programas se les llama shader programs, y el desarrollador debe escribir una función principal que reciba y retorne una serie de parámetros, también definidos por el programador dentro de ciertos límites. En lo que resta de esta tesina, se utilizan los términos etapas programables y shaders de forma indistinta.

Los shaders son construidos sobre un conjunto de funcionalidades en común a todos los shaders, llamado common shader core. El common shader core define el diseño de entrada y salida de los datos utilizado en la etapa programable, proveyendo funciones intrínsecas disponibles en todas las etapas programables.

Durante la ejecución de los shaders, además de tener a su disposición los datos recibidos por parámetro, como se muestra en la figura 33 también tienen acceso a los recursos en memoria, cuya interfaz a ellos también es provista por el common shader core. Nótese que los recursos a los que pueden acceder los shaders son los mismos sobre los que opera el compute shader (kernel de DirectCompute) utilizado durante la etapa de simulación.

En la figura se muestra el diseño definido por el common shader core que implementan todas las etapas programables.

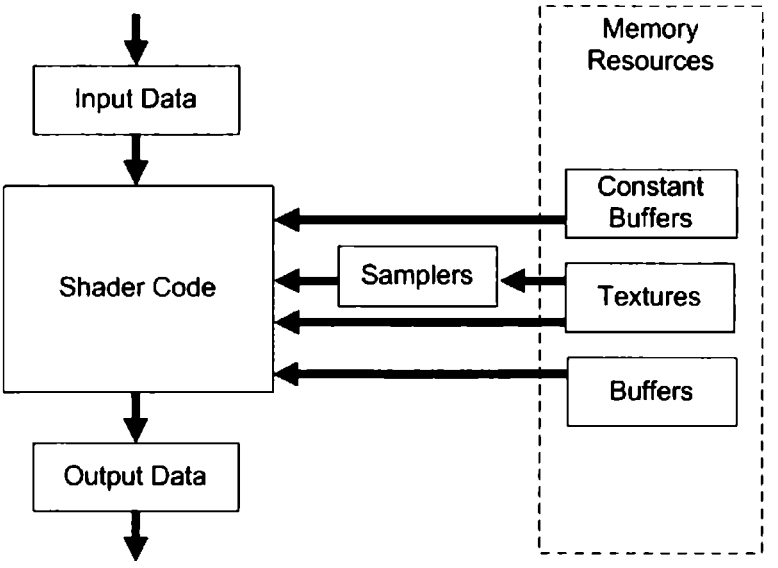


Figura 33: Common shader core. ***[32]***

Como se mencionó anteriormente, los datos de entrada y de salida en el shader son definidos por el programador, sin embargo cada etapa posee distintos tipos de datos de entrada y de salida que deberán mantenerse. Por ejemplo en el Vertex Shader se requiere

que en los datos de salida exista un atributo representando la posición de un vértice, la cual será utilizada por el Rasterizer en la siguiente etapa.

Además de la funcionalidad en común que comparten todos los shaders, cada etapa provee funcionalidad especial que sólo puede ser utilizada en la misma. El comportamiento individual de cada etapa será discutido en la siguientes secciones.

En conclusión, las etapas fijas junto a las programables componen el pipeline que permite realizar el renderizado de imágenes. El proceso de ejecución del pipeline consiste en configurar todas las etapas a ser utilizadas, realizar la asignación de recursos (resource binding) de entrada que son utilizados en cada etapa, así como también los recursos de salida del pipeline, y por último llamar a una función de la API de Direct3D (Draw, DrawIndexed, etc) para comenzar la ejecución propiamente dicha.

El resultado de la ejecución es típicamente una imagen en forma de textura. Múltiples ejecuciones pueden llevarse a cabo reutilizando la misma textura como recurso de salida, dando como resultado la acumulación de los resultados en cada ejecución. Por ejemplo si se desea renderizar tres objetos en pantalla en la misma imagen (en las siguientes secciones se entrará más en detalle), se realizarán tres ejecuciones del pipeline utilizando la misma textura para almacenar los resultados. Esto implica que podrían utilizarse distintos shaders para el renderizado de cada objeto, ya que cada uno se da en una ejecución distinta del pipeline.

En las secciones restantes se entra en detalle a cerca del funcionamiento de cada etapa, haciendo mención de los tipos de datos que se utilizan y cómo son transformados.

8.2.2 Etapas del *Rendering Pipeline*

Antes de comenzar, primero se da una vista general del funcionamiento del pipeline para luego ayudar a comprender cómo las etapas se relacionan entre ellas y que tipo de operaciones realiza cada una.

El proceso de renderización comienza con el host enviando vértices (y otros recursos) como datos de entrada a la GPU. Los vértices son utilizados para conformar triángulos, y a su vez éstos triángulos componen las figuras geométricas que se desean renderizar. El triángulo es la unidad utilizada por la GPU para realizar la rasterización de las superficies. Esto se debe a que, por un lado, el triángulo otorga un alto nivel de simplicidad al momento de realizar las operaciones matemáticas necesarias durante el proceso de rasterización, y por otro lado cualquier superficie geométrica puede ser descompuesta o subdividida en triángulos. Por ejemplo un cuadrado puede ser subdividido en dos triángulos, como se muestra en la siguiente figura:

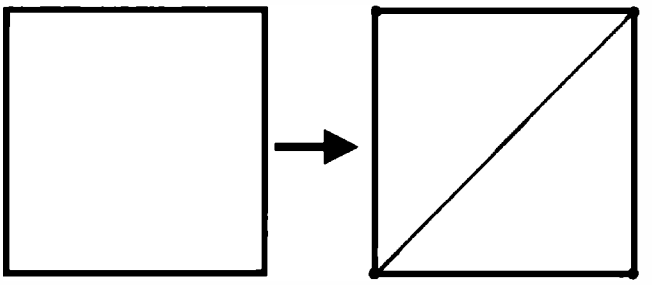


Figura 34: Subdivisión de un cuadrado en dos triángulos.

Los vértices pueden contener varios atributos definidos por el desarrollador según los necesite, pero como mínimo se deberá proveer una posición en el espacio, utilizando como estructura de dato un vector de 3 o 4 componentes en punto flotante[28].

Una vez finalizado el proceso de ensamblado de los vértices, se inicia la ejecución del vertex shader pasando como parámetro los datos de dicho vértice, realizándose una invocación del vertex shader por cada vértice que haya. Es importante mencionar que la GPU no provee ninguna garantía con respecto al momento en el que se realizarán las invocaciones del vertex shader. Por ejemplo si se deben procesar 50 vértices, la GPU podría realizar las primeras 20 ejecuciones en paralelo y luego las restantes 30, o una vez terminada las primeras 20 podría enviar los datos a las siguientes etapas y mas tarde procesar los 30 vertices pendientes.

Una vez el vertex shader termina su ejecución, los resultados pasan a la etapa de rasterización, la cual consiste (entre otras cosas) en determinar qué píxeles en pantalla son cubiertos por la superficie de cada triángulo, para luego realizar una invocación del fragment shader por cada pixel ocupado en pantalla. En la siguiente figura se encuentra ilustrado el proceso.

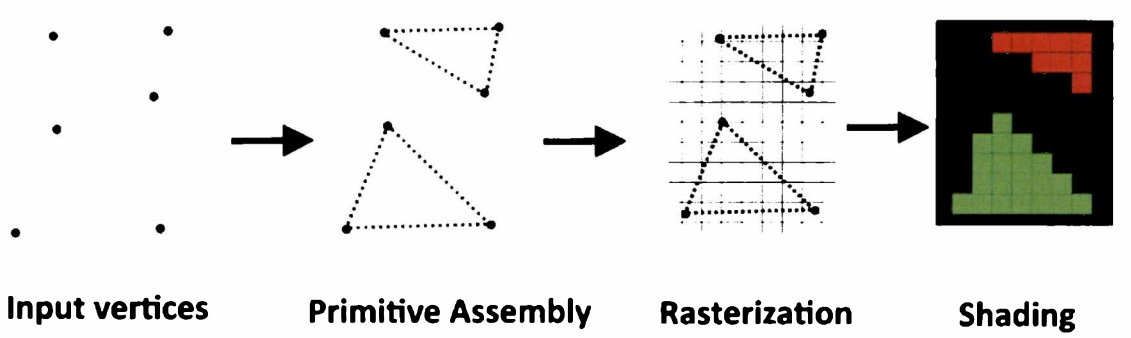


Figura 35: Etapas del rendering pipeline.

El resultado de la ejecución de un fragment shader es el color del pixel que será mostrado en pantalla o almacenado en una textura.

[28] Ver más adelante el motivo por el cual se utilizan 4 elementos.

Input Assembler

El Input Assembler es la primera etapa en el pipeline y es la encargada de construir los vértices que serán procesados en las siguientes etapas. Para esto, el host debe proveer una serie de buffers los cuales contienen cada uno de los atributos que componen un vértice. A estos buffers se los llama Vertex Buffer Object (VBO), y el desarrollador puede elegir entre almacenar varios atributos en un mismo VBO o utilizar un VBO por atributo.

Además de los datos a almacenar, al VBO se le debe asociar una etiqueta llamada binding semantic, la cual es utilizada más adelante por el vertex shader para poder acceder al buffer. Esta etiqueta no es más que un simple identificador.

El Input Assembler utiliza todos los VBOs para construir cada uno de los vértices, y también es el encargado de determinar cómo los vértices se conectan entre ellos para formar lo que se conoce como primitiva geométrica. Existen tres tipos de primitivas[29]: puntos, líneas y triángulos. Las primitivas componen la superficie de la geometría que se desea renderizar, y dependiendo del tipo elegido, el resultado final puede variar drásticamente. Por ejemplo, si una superficie geométrica está compuesta por triángulos pero el tipo de primitiva fue configurado como puntos, el resultado será una nube de puntos en lugar de una superficie, con varios de los puntos posiblemente superpuestos entre ellos.

Además de especificar el tipo de primitiva, el Input Assembler también debe poseer la información necesaria para poder interpretar correctamente los vértices que componen la primitiva utilizada. Si en un VBO se almacenan las posiciones de cuatro vértices y se utilizan líneas como primitiva, el Input Assembler aún no tiene manera de saber cuantas lineas hay en el buffer. Esta información es provista dependiendo del modo que se utilice para la ejecución del pipeline. Por ejemplo, en el caso de que se utilicen cuatro vértices (**v1, v2, v3**,**v4**), al invocar el pipeline utilizando el método **Draw** el InputAssembler irá creando las líneas utilizando los vértices en el orden de aparición en el buffer, dando como resultado 2 líneas, la primera conformada por los vértices **v1** y **v2**, y la segunda conformada por los vértices **v3** y **v4**.

Figura 36: Dos líneas compuestas por dos vértices cada una.

Si se comienza la ejecución utilizando el método **DrawIndexed**, se puede utilizar un buffer especial llamado Index Buffer Object (IBO), el cual contiene índices (números enteros) indicando los vértices que componen cada una de las líneas. Continuando con el ejemplo anterior de cuatro vértices, para realizar una línea continua el IBO puede contener los siguientes valores:

[29] Existe un cuarto tipo llamado *quad* utilizado en etapas de teselado que no serán discutidas en esta tesina.

```
IBO = new int[] {0, 1, 1, 2, 2, 3};
```

Dando como resultado tres líneas en lugar de dos, como se muestra en la siguiente figura:

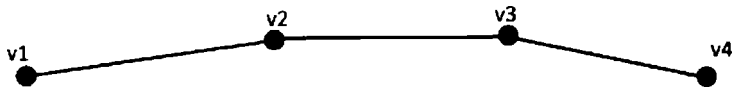


Figura 37: Tres líneas compuestas por cuatro vértices.

Esto permite reducir enormemente la cantidad de memoria utilizada al reutilizar un mismo vértice para más de una primitiva; en este caso un vértice es utilizado para el final de una línea y el comienzo de otra. Si se quisiese lograr tres líneas interconectadas (al igual que en el ejemplo anterior) utilizando el método **Draw**, se deberán utilizar vértices extra posicionados al final de cada una de los segmentos, para así utilizarlos como punto de partida del nuevo segmento:

Figura 38: Tres líneas compuestas por seis vértices.

En conclusión, el Input Assembler forma la conexión primaria entre los datos de la aplicación y el rendering pipeline. Se encarga de generar los vértices utilizando una serie de buffers conteniendo los atributos necesarios que compondrán a cada vértice, así como también el tipo de primitiva e información acerca de cómo los vértices se encuentran interconectados entre ellos. Una vez realizado esto, comienzan las invocaciones de los vertex shader pasando los vértices resultantes como parámetro.

Vertex shader

El vertex shader es la primer etapa programable que se encuentra en el rendering pipeline. El host deberá compilar un programa escrito en HLSL y asociarlo al pipeline para ser ejecutado. El programa estará compuesto por una función principal que recibirá como parámetro uno de los vértices ensamblados en el Input Assembler, y retornará uno nuevo posiblemente con una cantidad distinta de atributos.

El vertex shader no tiene noción del tipo de primitiva con la que se está trabajando, y cada invocación se hace de forma completamente aislada y sin ningún tipo de comunicación con las otras instancias. Su único propósito es el de transformar los datos del vértice para pasarlos a la etapa de rasterización. El resultado de cada invocación es combinado nuevamente con la información de la topología (primitiva geométrica) para ser utilizado por las siguientes etapas en el pipeline. Todo esto le permite al vertex shader operar sobre

todos los vértices de la misma manera, independientemente del tipo de primitiva con la que se trabaje.

El programa del vertex shader, escrito en HLSL, debe declarar una estructura con los atributos que recibe del Input Assembler, y es fundamental que ambos atributos coincidan. Es decir, el vertex shader debe consumir todos los atributos que posea cada vértice. Esto se realiza mediante la etiqueta mencionada anteriormente. A continuación se da un ejemplo de declaración de la estructura y la función del shader:

Estructura y declaración de un vertex shader

```
struct vertex_shader_input{
      float3 pos : POSITION;
}

struct vertex_shader_output{
      float3 pos : SV_POSITION;
}

vertex_shader_output main_VS (in vertex_shader_input i){
      vertex_shader_output o;
      o.pos = i.pos + float3(0,1,0);
      return o;
}
```

En este ejemplo se puede observar que en la estructura **vertex_shader_input** se encuentra declarada la variable **pos**, asociada a la etiqueta **POSITION**. Esto quiere decir que debe haber al menos un VBO enviado al Input Assembler que contenga las posiciones de los vértices y esté asociado a dicha etiqueta. La función **main** realiza una transformación trivial de los datos, y los almacena en la estructura de salida (**vertex_shader_output**) la cual posee una variable **pos** con una etiqueta especial. Previamente se mencionó que algunas etapas tendrían sus datos de entrada o salida condicionados por lo que se requiera en la etapa siguiente (o previa). En este caso, la siguiente etapa es la rasterización, y en ella se necesita que entre los atributos de los vértices de salida exista uno con la etiqueta **SV_POSITION**[30] representando la posición del vértice en el espacio tridimensional. Esta posición es utilizada por el rasterizer para proyectar la posición en la pantalla a la que se corresponde cada vértice, y rasterizar la superficie del triángulo que conforman los vértices (se dará mas información en la sección de rasterización).

Si bien es necesario que exista un atributo con la etiqueta **SV_POSITION** en los datos de salida, no es necesario que lo mismo se cumpla para los atributos de entrada. Es decir, no

[30] **SV** proviene de **System Value**, y representan semánticas especiales declaradas por el sistema.

es obligatorio que los vértices de entrada posean un atributo con la etiqueta **POSITION** o **SV_POSITION.**

También es importante mencionar que el vertex shader tiene dos fuentes de recursos de los cuales puede recibir datos. Por un lado están los atributos de cada vértice recibidos por parámetro mediante el Input Assembler, como por ejemplo la posición. Por otro lado, existen los recursos externos asignados por el host como parte de la configuración de la etapa. Por ejemplo se podrían asignar texturas o constantes que el vertex shader puede utilizar para realizar las transformaciones de los datos de salida.

Por último, algunas de las operaciones más comunes que se suelen realizar en el vertex shader, son:

• Multiplicación de matrices sobre la posición de los vértices, para desplazar o transformar de alguna forma la geometría en el espacio.
• Calcular la iluminación en cada vértice. La iluminación por vértice es una técnica vieja que hoy en dia es poco utilizada, dado que existen los recursos suficientes como para calcular la iluminación a nivel de fragmento, algo que hace 15 años implicaba un alto costo de cómputo.
• Animaciones basadas en esqueletos (también llamado vertex skinning).

Rasterizer

El rasterizer es la última etapa en el pipeline que lidia estrictamente con datos geométricos. El principal propósito de esta etapa es el de convertir primitivas a una representación muestreada regularmente que luego permita almacenar su información en lo que se conoce como render target, ubicado al final del pipeline. A este proceso de muestreo se le llama rasterización, y da como resultado la creación de **fragmentos** que intentan aproximar la forma de la primitiva al ser proyectada sobre la pantalla. Como se puede observar en la siguiente figura, el rasterizer recibe vértices como entrada y genera múltiples fragmentos de salida, funcionando como etapa de amplificación de datos.

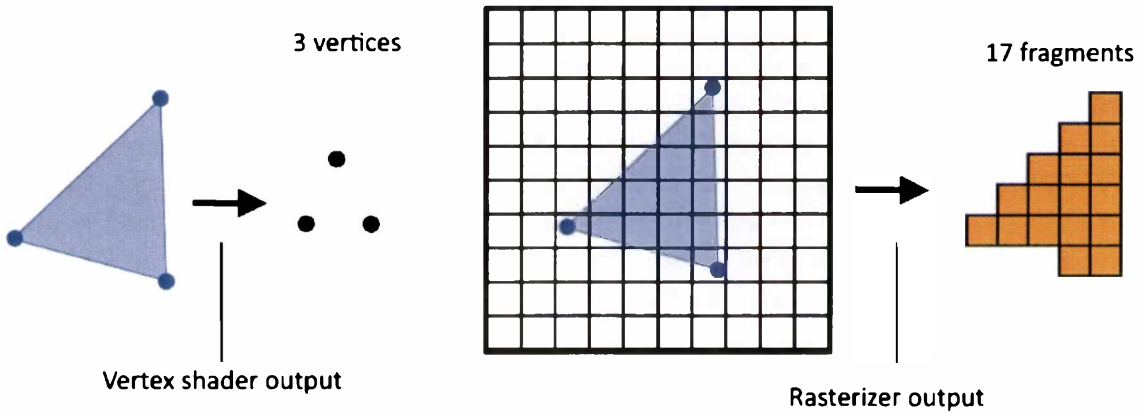


Figura 39: Amplificación de datos durante la rasterización.

Es por esto que antes del proceso de rasterización, el rasterizer realizará dos operaciones llamadas primitive culling y primitive clipping destinadas a reducir la cantidad de fragmentos generados tanto como sea posible.

La primera operación a realizar es la de culling, y se encarga de eliminar aquellas primitivas que no contribuirán en lo absoluto al renderizado de la imagen final. Hay dos tipos de *culling* que se deben realizar: ***primitive culling***, y ***backface culling***.

Antes de continuar, es necesario hacer una breve mención del concepto de cámara durante el renderizado de una escena, y las distintas transformaciones que sufren los vértices antes de llegar a la rasterización.

El rasterizer no maneja el concepto de cámara, pero mediante el uso de matrices para realizar transformaciones de espacio sobre las posiciones de los vértices, es posible crear una cámara virtual que permita al usuario navegar por la escena. Para esto en el vertex shader se aplican diversas transformaciones, siendo la mas importante la transformación de proyección la cual modifica la posición de los vértices basándose en el view frustum de la cámara virtual, ubicándolos en lo que se conoce como clip space o projective space. Luego, el rasterizer aplica una última transformación a estas posiciones, llamada división de perspectiva, haciendo que las posiciones de los vértices queden en el rango -1..1 en el eje X e Y, y 0..1 en el eje Z. A todo el espacio contenido dentro de dicho rango, se le llama clipping volume, y al espacio de coordenadas se le llama normalized device coordinates.

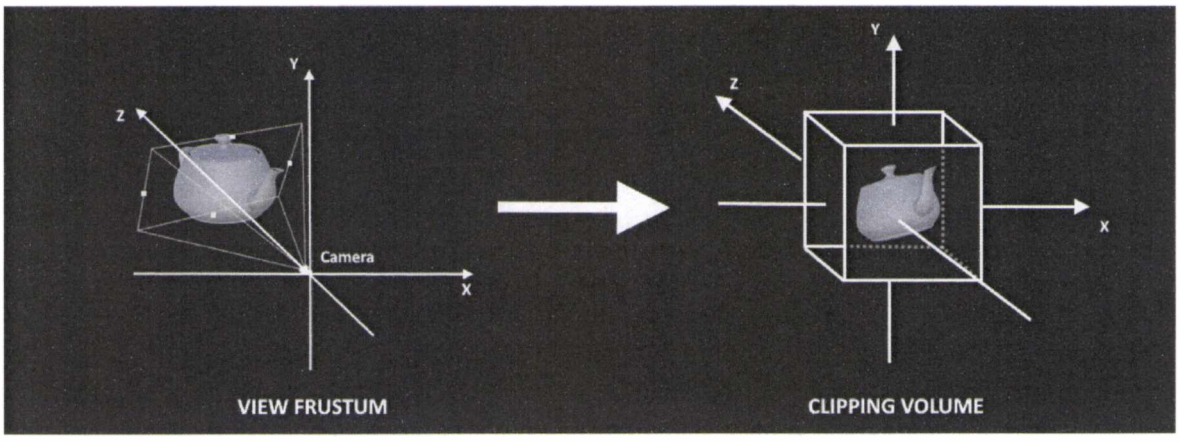


Figura 40: Transformación de espacios de coordenadas.

En la operación de **primitive culling** se eliminan todas aquellas primitivas que estén fuera del clipping volume en su totalidad, es decir, fuera de cámara. Por ejemplo si un triángulo posee los tres vértices fuera del clipping volume, será descartado y no llegará al proceso de rasterización, ilustrado en la siguiente figura:

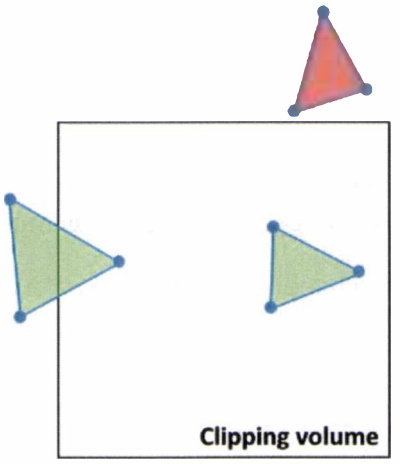


Figura 41: *Primitive culling. El triángulo rojo será descartado.*

La operación de **backface culling** consiste en descartar a los triángulos dependiendo de la orientación de su cara. La cara de un triángulo es calculada utilizando el producto vectorial entre los bordes (vectores) que parten desde el vértice inicial al siguiente, y entre el vértice inicial y al último, como se ilustra en la siguiente imagen.

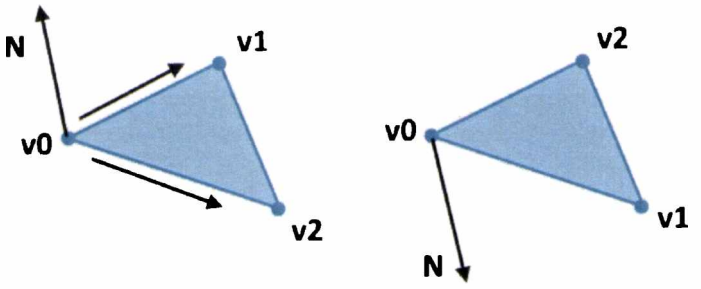


Figura 42: *Relación entre el vector normal del triángulo y el orden de los vértices.*

Si la orientación de la cara apunta hacia la cámara, entonces el triángulo sobrevive, de lo contrario es descartado. Esta operación es útil ya que generalmente cuando se poseen geometrías "cerradas" como por ejemplo la de un cubo, las caras opuestas a la cámara no son visibles, y por lo tanto no hace falta que pasen por el proceso de rasterización.

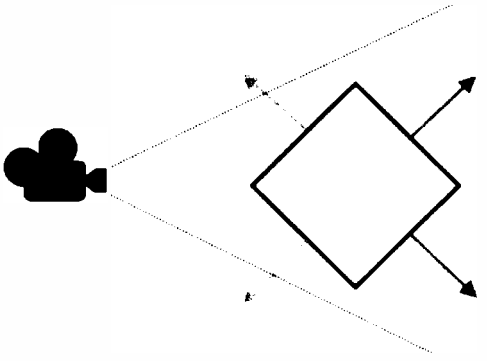


Figura 43: Backface culling

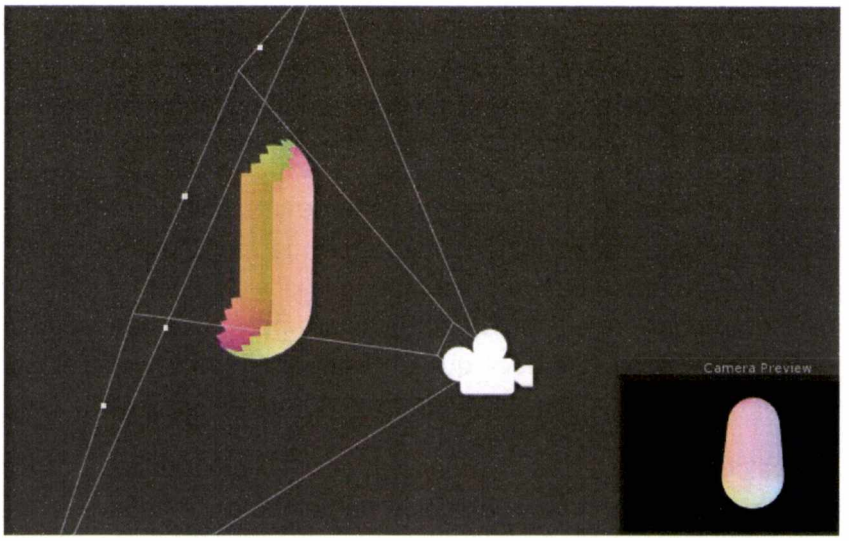


Figura 44: Backface Culling

Sin embargo existen casos en los que es preferible permitir que se preserve el triángulo independientemente de la orientación de sus caras, por ejemplo si la geometría que se desea renderizar es una hoja de papel que debe ser vista de ambos lados[31]. Para esto, el rasterizer permite que se pueda configurar la forma en la que se realiza el descarte del triángulo. Los modos a configurar son: backface culling, front face culling, y no culling. En el caso de la hoja de papel, se debe deshabilitar el culling de las caras.

Una vez finalizada la etapa de primitive culling, el resultado se compone por todas aquellas primitivas que se encuentren total o parcialmente dentro del clipping volume. Luego se procede a la etapa de primitive clipping, en donde se toman todas las primitivas que hayan sobrevivido a la etapa de culling y se buscan aquellas que se encuentren parcialmente fuera del clipping volume, y se las divide en nuevas primitivas que se encuentren dentro del clipping volume en su totalidad, descartando las que queden afuera. El proceso se ilustra en la siguiente imagen:

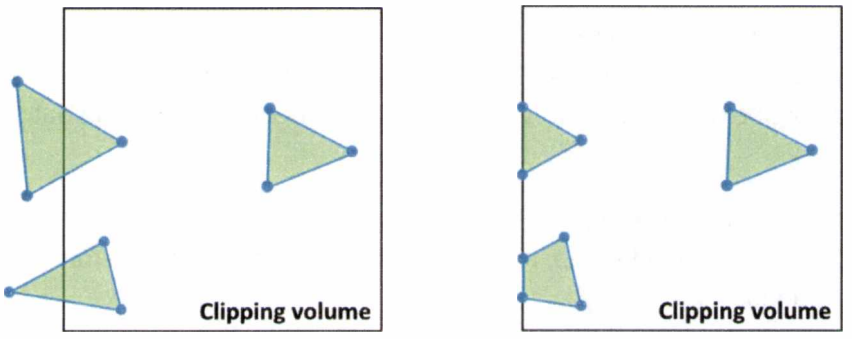


Figura 45: *Proceso de primitive clipping.*

[31] En este ejemplo se asume que se utiliza un plano para la hoja de papel, aunque también se podría utilizar un cubo rectangular extremadamente delgado para representar el grosor del papel, pero este tipo de prácticas es muy poco común.

Una vez finalizadas las etapas de culling y clipping, se sabe que todas las primitivas resultantes se encuentran dentro del clipping volume. Luego, las siguientes etapas del rasterizer se encargarán de preparar las primitivas y sus atributos para el proceso de rasterización.

La última etapa que de interés en esta tesina, es la de interpolación de atributos para cada uno de los fragmentos generados. Anteriormente se mencionó que el vertex shader debe retornar obligatoriamente un atributo representando la posición del vértice en pantalla, pero también es posible agregar otros tipos de atributos al vértice para que luego sean utilizados en el fragment shader. Esto quiere decir que el rasterizer debe calcular el valor del atributo que le corresponde a cada uno de los fragmentos generados.

Para realizar el cálculo, el rasterizer interpola el valor de cada uno de los atributos de los vértices que componen la primitiva. El valor del atributo contribuye al resultado final basándose en la distancia que existe entre el fragmento generado y los vértices. Cuanto más cerca esté el fragmento de uno de los vértices, más influencia tendrá ese vértice sobre la interpolación de los atributos.

Por ejemplo en la siguiente imagen se puede ver el caso de la rasterización de una línea, en donde se poseen dos vértices (**v1, v2**) a los cuales el programador les asoció un atributo que representa el color en cada vértice. Aquellos fragmentos ubicados exactamente en la misma posición del vértice (**f0, f5**) tomarán el 100% del color que posee dicho vértice. Todos los fragmentos intermedios (**f1..f4**) obtendrán los colores interpolados entre un vértice y otro.

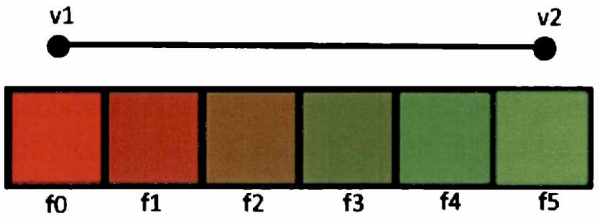


Figura 46: Interpolación de los colores de los vértices v1 y v2, a lo largo de los fragmentos f0..f5

La misma situación se da para el caso del triángulo. El concepto de interpolación de atributos es fundamental ya que es el mecanismo por el cual los datos del vertex shader se transfieren al fragment shader para poder ser utilizados en el procesamiento de la imagen final.

Una vez finalizada la rasterización, el resultado son todos los fragmentos en donde cada uno posee la misma cantidad de atributos que el vertex shader retornó, pero con sus valores interpolados.

Fragment Shader

El fragment shader funciona de manera similar al vertex shader. Es una etapa programable, también escrita en HLSL, y se ejecutará una instancia por cada fragmento que haya sido

generado por el rasterizer. Su ejecución se hace de forma aislada a las otras instancias, de forma tal que cada instancia no tiene ningún tipo de comunicación con las demás.

En esta etapa se debe tener cuidado con la cantidad de operaciones y el tipo de operaciones que se realizan, ya que a diferencia de los vértices, la cantidad de ejecuciones del fragment shader es considerablemente mayor. Por ejemplo si se debe rasterizar un rectángulo (compuesto de dos triángulos) que ocupa toda la pantalla, y la resolución de la pantalla[32] es de 1920x1080, entonces habrá 2.073.600 ejecuciones del fragment shader, y solo 4 ejecuciones del vertex shader. Si en cada fragment shader se accede múltiples veces a memoria, muy fácilmente se puede llegar al límite del ancho de banda disponible, degradándose la performance de la aplicación.

El fragment shader recibe cada uno de los fragmentos generados por el rasterizer, con la misma estructura de datos que retornó el vertex shader. Debe retornar únicamente un color, representado por arreglo de cuatro componentes (RGBA) en el rango 0..1 en punto flotante, asociado a la etiqueta **SV_TARGET**. A continuación se muestra un ejemplo en el cual el fragment shader recibe la misma estructura declarada en el vertex shader, y retorna el color rojo:

```
float4 main_FS (in vertex_shader_output i) : SV_TARGET{
      return float4(1,0,0,0); // (Red, Green, Blue, Alpha)
}
```

Nótese que en este caso, la etiqueta **SV_TARGET** se declara junto con el método. Si se utiliza este fragment shader para renderizar una esfera compuesta de triángulos, el resultado será una esfera totalmente roja. Por lo tanto, el fragment shader es la etapa responsable de darle a la primitiva geométrica su apariencia visual.

También se debe mencionar que una vez que la posición del fragmento fue determinada por el rasterizer, no es posible modificarla.

Al igual que en el vertex shader, si se desea se pueden asociar distintos recursos a ser utilizados en el fragment shader, como por ejemplo texturas o constantes. Estos recursos deberán ser asociados por el host antes de que comience la ejecución del pipeline.

Output Merger

El output merger es la etapa final en el pipeline, y se encarga principalmente de recibir los resultados de cada uno de los fragmentos y combinarlos con los colores ya almacenados en el render target. El render target es similar a una textura pero con funcionalidades especiales, y son utilizados para mostrar los contenidos en la pantalla, entre otras cosas. Durante la rasterización serán las dimensiones del render target las que determinan qué

[32] Esto no depende de la resolución de la pantalla, sino de la resolución del *render target*, del cual se hablará en la sección del Output Merger

partes de la primitiva son cubiertas por un pixel. Es decir, si se utiliza un render target con una resolución de 512x512, y se desea renderizar un rectángulo que cubra toda la pantalla, se generarán 512x512 fragmentos independientemente de la resolución de la pantalla (la cual podría ser de 1920x1080).

Además de almacenar los resultados en el render target, el Output Merger provee dos funcionalidades extra.

La primera permite modificar el color resultante del fragmento antes de ser almacenado, mediante lo que se conoce como blending mode. Si se encuentra activado (por el desarrollador), se puede elegir cómo modificar el color del fragmento en base al color existente del render target en donde éste será almacenado. Por ejemplo uno de los modos a elegir es el additive blending en donde el color resultante del fragmento, llamado source color, es sumado al color existente en el render target, llamado destination color. Existen varios modos para configurar, siendo el más común el alpha blending, el cual permite el uso de transparencia.

La segunda funcionalidad importante que ofrece el Output Merger es la de realizar pruebas de visibilidad mediante Z-Testing para descartar aquellos fragmentos que tengan una profundidad mayor al ya existente en el buffer de profundidad. Luego de la rasterización, el rasterizer además de interpolar los atributos enviados por el vertex shader también interpola la posición, y utiliza el eje Z para determinar la profundidad que posee el fragmento. Se debe recordar que la posición se encuentra en normalized device coordinates, y por lo tanto el rango de valores del componente Z es de 0..1. Cuanto menor sea el número, menor profundidad tendrá con respecto a la cámara. Esta profundidad es asociada al fragmento y utilizada por el Output Merger para determinar si un fragmento se encuentra por "debajo" de otro al momento de ser almacenado en el render target.

Por ejemplo si primero se renderiza un triángulo cuyos vértices (en normalized device coordinates) tienen una profundidad de 0.5, y luego se renderiza otro en exactamente la misma posición pero con una profundidad de 0.2, entonces todos los fragmentos que genere el nuevo triángulo reemplazarán los ya existentes en el render target. Sin embargo, si los vértices del segundo triángulo poseen una profundidad de 0.7, se interpreta que están por detrás del primer triángulo, y por lo tanto los fragmentos serán descartados sin ser almacenados en el render target. En este ejemplo, se utilizó una función de comparación en la cual el fragmento será descartado si posee una profundidad mayor a la existente en el render target. A esta comparación se le llama Z-Testing, y puede ser configurada para realizar el descarte de distinta forma, o incluso ser desactivado por completo. Es posible desactivar el Z-Testing y la escritura en el buffer de profundidad, haciendo que no sea necesario asociar un buffer de profundidad al Output Merger.

Una vez que finaliza la ejecución del pipeline, el render target podrá ser utilizado para mostrar los resultados en pantalla, o puede ser utilizado en una nueva ejecución del pipeline sin descartar sus contenidos, para realizar así la composición de todos los objetos en la escena.

Con esto se finaliza la introducción del rendering pipeline. Se debe recordar que se ha presentado una visión simplificada del mismo y se han omitido intencionalmente varios detalles que si bien son importantes, no son necesarios para comprender la implementación realizada en esta tesina.

8.2.3 Implementación de la visualización

Para la implementación de la visualización se utiliza la API que provee Unity para interactuar con el subsistema de Direct3D y sus recursos. El objetivo es visualizar la densidad del humo sobre un plano ubicado en la escena, y con cierto grado de transparencia. A continuación se presenta el código y la configuración relevante de cada una de las etapas.

Primero se crean los datos de los vértices que serán enviados al Input Assembler. Para esto se utiliza la clase Mesh provista por Unity, la cual contiene la información de los atributos de cada vértice, así como también los índices y topología a utilizar.

Configuración del Input Assembler

```
Mesh mesh = new Mesh ();
List<Vector3> vertices = new List<Vector3> ();
vertices.Add (new Vector3(-1, -1, 0));
vertices.Add (new Vector3( 1,  1, 0));
vertices.Add (new Vector3( 1, -1, 0));
vertices.Add (new Vector3(-1,  1, 0));
mesh .SetVertices (vertices);
int[] indices = new int[]{0,1,2,  0,3,1};
mesh .SetIndices (indices, MeshTopology.Triangles, 0);

List<Vector2> uvs = new List<Vector2> ();
uvs.Add(new Vector2(0, 0));
uvs.Add(new Vector2(1, 1));
uvs.Add(new Vector2(1, 0));
uvs.Add(new Vector2(0, 1));
mesh .SetUVs (0, uvs);
mesh .UploadMeshData ();
```

Como se puede observar, primero se crean las cuatro posiciones de los vértices que compondrán al plano, cada posición representa una de las esquinas. Luego, se crean los índices que indican los dos triángulos a utilizar. El primer triángulo está compuestos por los vértices 0, 1 y 2, y el segundo triángulo por los vértices 0, 3 y 1. Se debe tener especial cuidado en el orden en el cual se indican los índices, ya que dependerán del orden en el cual fueron introducidos los vértices. La geometría resultante se puede apreciar en la siguiente ilustración:

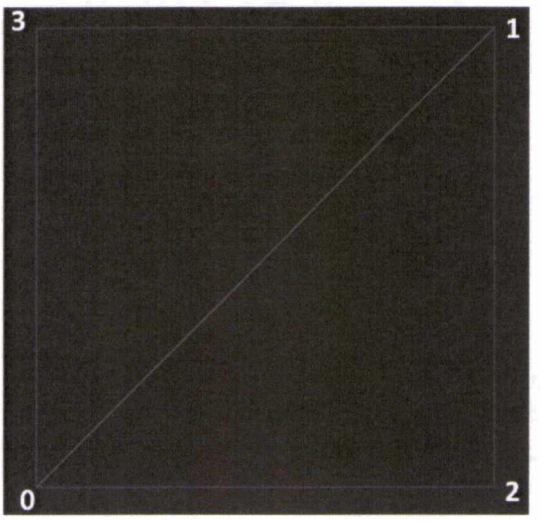


Figura 47: Geometría resultante, compuesta por dos triángulos.

Nótese que al asignar los índices utilizando el método **SetIndices** también se indica el tipo de primitivas a utilizar, en este caso triángulos, para que el Input Assembler pueda interpretar los datos correctamente.

Por último, se agrega un atributo adicional a los vértices, llamado **uvs**. Su uso será detallado más adelante.

La clase Mesh automáticamente se encarga de asociar cada uno de los atributos al InputAssembler, y asignar las etiquetas correspondientes. En el caso de las posiciones se asigna la etiqueta **POSITION**, y en el caso de las uvs se asigna la etiqueta **TEXCOORD0**.

Una vez configurado el Input Assembler, se configuran los programas a utilizar en el vertex y fragment shader. Para esto Unity utiliza la clase Material, la cual contiene no solo los shaders, sino también los recursos que utilizará cada uno. Primero se crea el material pasando como parámetro la ubicación del archivo que contiene el código ambos programas, y luego se asigna al material la textura de densidad del humo que se obtuvo como resultado de la simulación.

Configuración de vertex y fragment shader

```
Material mat = new Material (Shader.Find ("shaders/visualize_smoke"));
mat.SetTexture ("_Density", density_tex_0);
```

Por una cuestión de conveniencia y simplicidad, Unity permite que tanto el vertex shader como el fragment shader puedan ser declarados en el mismo archivo, agregando una sintaxis especial para configurar varios otros aspectos del rendering pipeline. En este archivo se activa el backface culling para descartar los triángulos cuya cara no pueda ser vista por la cámara.

Una vez configurados los programas y recursos a utilizar, solo queda la configuración del output merger. Para esto se procede a crear el render target (llamado RenderTexture) en el que se guardarán los resultados del renderizado, y luego se lo asigna al output merger:

Configuración del Output Merger

```
RenderTexture render_target = new RenderTexture (Screen.width,
Screen.height, 24, RenderTextureFormat.ARGB32);

Graphics.SetRenderTarget(render_target);
```

Los primeros dos parámetros indican las dimensiones del render target, en este caso las dimensiones son las mismas que las de la resolución de la pantalla, para poder visualizar el resultado en la totalidad de la pantalla. El siguiente parámetro es la precisión que se quiere utilizar para el buffer de profundidad, en este caso 24 bits. Por último se indica el formato que poseerá el render target, en este caso se utilizan 8 bits para cada canal RGBA, permitiendo un rango de colores de 0 a 255 en cada uno.

En el archivo que contiene el vertex y fragment shader, se configura al output merger para escribir en el buffer de profundidad, se activa el Z-Test, y la función de comparación para el descarte de fragmentos será la de mayor o igual profundidad, de forma tal que se descarten todos los fragmentos que posean una profundidad mayor a la actual en el render target. También se configura el blending mode para que combine los colores basándose en la transparencia (canal Alpha) del color resultante del fragmento.

Por último una vez que haya sido configurado todo, solo resta iniciar la ejecución del pipeline con la siguiente instrucción:

Ejecución del pipeline

```
Graphics.DrawMesh(mesh, new Vector3(0, 0, 0), Quaternion.identity, mat);
```

La función DrawMesh internamente llama a la función DrawIndexed de Direct3D, la cual utiliza los índices provistos para construir las primitivas. El primer parámetro es el mesh, el segundo una posición en la escena, en este caso el centro. El tercer parámetro es la rotación del plano, el cual en este caso no tiene rotación. Por último, el material con los shaders y recursos.

La inicialización de la cámara virtual y el resto de las configuraciones son provistas por parte de Unity, utilizando configuraciones por defecto en los casos que no se especifiquen.

A continuación se presentan el vertex y fragment shader utilizados.

Vertex y Fragment shader

```
struct appdata{
```

```
    float4 position: POSITION;
    float2 uv : TEXCOORD0;
};
struct vertex_shader_output{
    float2 uv : TEXCOORD0;
    float4 position: SV_POSITION;
};

sampler2D Density;
vertex_shader_output vertex_shader (appdata v){
    vertex_shader_output o;
    o.position = UnityObjectToClipPos(v.position);
    o.uv = v.uv;
    return o;
}

float4 fragment_shader(in vertex_shader_output  i) : SV_Target{
    return tex2D( Density, i.uv);
}
```

El vertex shader recibe por parámetro una estructura con la posición del vértice y la coordenada uv, provistas por el Input Assembler. La posición es transformada de local space (también llamada object space) a clip space mediante una función de conveniencia que provee Unity llamada UnityObjectToClipPos. Internamente la función realiza una multiplicación entre la posición y una matriz que representa todas las transformaciones necesarias para llegar al clip space. Dicha matriz está compuesta por una combinación de los siguientes tres elementos **[47]**:

- Posición, rotación y escala del objeto a renderizar.
- Posición y rotación de la cámara virtual.
- Todos los parámetros utilizados para la configuración la cámara virtual, los cuales dependerán del tipo de proyección a utilizar (ortográfica o perspectiva).

Una vez realizada la transformación, se almacena la posición resultante en la estructura de salida, junto con la posición UV la cual será transferida sin modificar.

Luego, el fragment shader realiza un muestreo de la textura de densidad mediante la función **tex2D**, utilizando las coordenadas uv para especificar la posición de la textura que se desea muestrear, dando como resultado el color almacenado en la posición especificada.

Las coordenadas UV son un espacio de coordenadas de una, dos o tres dimensiones (utilizadas en las funciones **tex1D**, **tex2D** y **tex3D** respectivamente) en punto flotante en el rango 0..1 usualmente utilizadas para realizar el muestreo de texturas. La función de muestreo realiza un mapeo entre las coordenadas UV y las ubicaciones de los texeles de la textura, permitiendo realizar el muestreo independientemente de su resolución. Por ejemplo si se posee una textura con una resolución de 1024x512, basta con realizar el muestreo

utilizando las coordenadas uv **(0.5, 0.5)** para conseguir el texel central. Internamente la función convierte dichas coordenadas de la siguiente manera:

x = 0.5 * 1024, **y** = 0.5 * 512

Dando como resultado el texel en la posición **(512,256)** en el eje **x** e **y** respectivamente.

Durante la creación de atributos, a cada vértice se le asignó la coordenada UV equivalente a cada una de las esquinas del sistema de coordenadas, para luego durante la interpolación de atributos dejar que cada fragmento reciba las coordenadas intermedias que luego serán utilizadas durante el muestreo. La siguiente imagen ilustra el proceso:

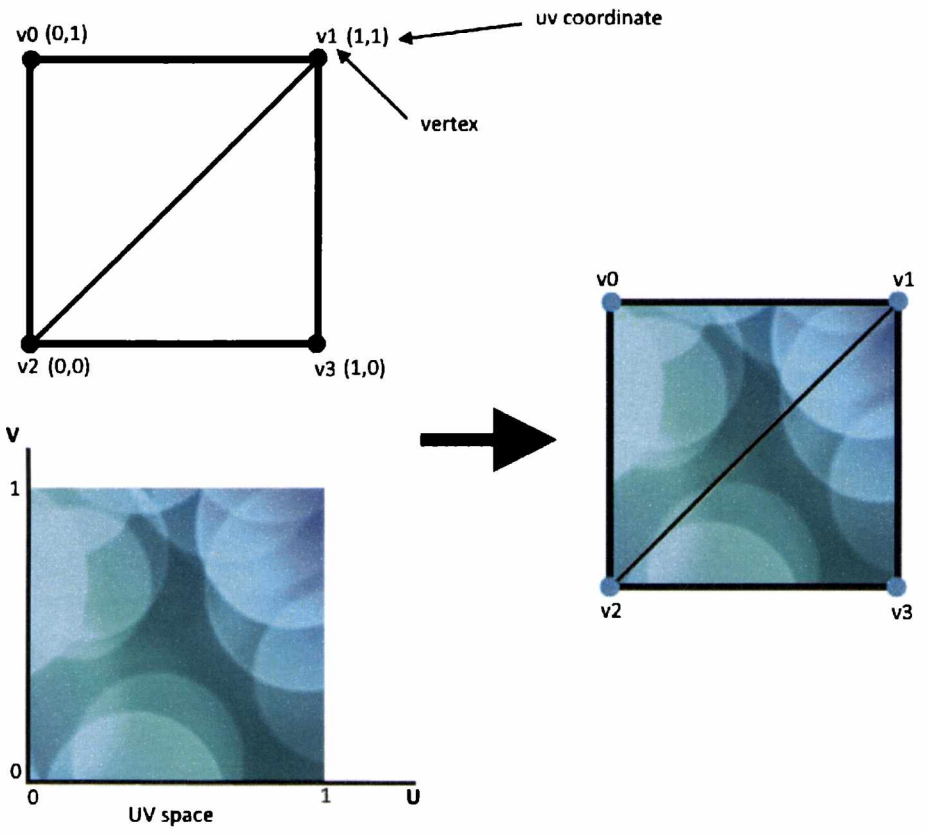


Figura 48: Mapeo de coordenadas UV de los vértices a una imagen.

Resta mencionar el uso del objeto **sampler2D** en el fragment shader. Este objeto determina cómo se debe realizar el muestreo en la textura, utilizando los filtros de minificación y magnificación, nivel de detalle y modos de direccionamiento de textura (el comportamiento que tiene el muestreo si las coordenadas UV son mayores a 1 o menores a 0). En este caso la textura de densidad utiliza la misma configuración que se utilizó durante la simulación de fluidos.

Con esto se concluye la implementación necesaria para poder visualizar los resultados de la simulación. Se presentaron las etapas vitales del pipeline gráfico junto a su configuración e inicialización, así como también el código a ejecutar sobre la GPU para conseguir los resultados esperados. En términos de rendimiento, las configuraciones y shaders utilizados son extremadamente básicos y no presentarán ningún tipo problema. Por lo tanto las optimizaciones que se deban realizar en este tipo de aplicaciones deberán estar orientadas a la etapa de la simulación, y no a la visualización.

9. CONCLUSIONES

En esta tesina se han presentado los conceptos físicos y matemáticos necesarios para poder realizar la implementación de una simulación de fluidos en tiempo real. Se siguió un método para obtener una simulación estable independientemente del timestep utilizado. con las tecnologías de DirectCompute y Direct3D se realizó una implementación en dos dimensiones que puso en práctica los conceptos desarrollados.

Se determinó que el principal problema de performance en este tipo de aplicaciones es el consumo de ancho de banda disponible en la GPU debido a la alta transferencia de datos generada por los kernels, particularmente el de la presión el cual debe realizar entre 30 y 50 ejecuciones para llegar a una convergencia aceptable. Una posible solución a este problema es hacer uso de la memoria compartida, la cual a su vez llevará a una reducción en el tiempo de ejecución de cada kernel. También, dependiendo del contexto en el que se utilice la simulación, se puede cambiar la precisión de las variables en punto flotante de 32 a 16 bits, lo cual lleva a una reducción del 50% en el consumo de ancho de banda y memoria almacenada.

Un problema adicional que se encuentra presente particularmente en las simulaciones de tres dimensiones, es el alto consumo de memoria, el cual puede llegar a los 8GB debido a las altas resoluciones de las texturas necesarias para poder mantener una buena calidad. Para remediar el alto consumo de memoria se sugirió el uso de una técnica llamada Sparse Grid Simulation**[29]**.

Para la visualización se presentó una versión del pipeline gráfico simplificado. Se detalló cada una de las etapas por las cuales los datos deben pasar, y cómo éstos son transformados. Se concluyó que la visualización en dos dimensiones será la etapa de menor consumo de recursos y poder de procesamiento, sin embargo en tres dimensiones se deberá recurrir a la utilización de técnicas de raymarching para el renderizado volumétrico, generando nuevamente problemas de performance con respecto al consumo de ancho de banda que deberán ser resueltos.

En la industria cinematográfica la simulación de fluidos en tiempo real puede acelerar enormemente el proceso iterativo involucrado en la creación de efectos especiales. En el área científica puede ser utilizado para visualizar un sistema de drenaje de agua, o el desplazamiento del agua provocado por un terremoto. En los videojuegos puede ser utilizado para agregar un nivel de realismo que no es posible de alcanzar utilizando las técnicas tradicionales para visualizar fluidos.

Hoy en día la simulación de fluidos en tiempo real es un área que está en constante crecimiento, y los métodos tradicionales utilizados para la simulación (los cuales resuelven las ecuaciones de Navier-Stokes o Euler) se comienzan a dejar de lado en favor de la utilización de otros métodos como los de Lattice-Boltzmann**[33]**, los cuales son altamente paralelizables y son particularmente buenos lidiando con fronteras de alta complejidad.

10. TRABAJOS FUTUROS

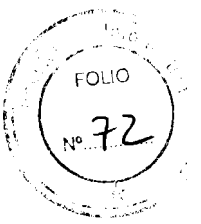
El programa desarrollado en esta tesina posee varios aspectos que podrían mejorarse, tanto en los algoritmos utilizados para la simulación, como en las optimizaciones de performance que se pueden realizar.

Para aumentar el grado de calidad general del fluido, se podría utilizar un algoritmo de advección llamado MacCormack Advection**[34]**, el cual provee un comportamiento más detallado, y con menor disipación de energía. Para un cómputo mas rápido de la presión se podrían utilizar los métodos llamados Conjugate-Gradient o Multigrid Methods**[35]**, ambos métodos poseen una convergencia mas rapida que el método iterativo de Jacobi, reduciendo enormemente la cantidad de memoria que debe ser transferida. También se podría cambiar el tipo de grilla utilizado, reemplazando la cell-centered grid por la staggered grid, aumentando la precisión en los resultados.

Los movimientos naturales del aire, humo y otros fluidos de baja viscosidad, generalmente contienen movimientos rotacionales llamados vórtices. Al discretizar el fluido en una grilla, inevitablemente se pierde una gran cantidad de los detalles provocados por los vórtices. Para restaurar una parte de estos detalles se podría utilizar un método llamado *vorticity confinement,* presentado por Fedkiw et. al **[11]**, el cual "inyecta" fuerzas rotacionales en el fluido.

En lo que respecta a las cuestiones técnicas de implementación, se podría aprovechar la memoria compartida de la GPU para acelerar enormemente el tiempo que tarda en ejecutarse cada uno de los kernels al reducir el consumo de memoria.

Si se desease convertir la simulación a tres dimensiones, se deberá utilizar un algoritmo de raymarching para poder visualizar el volumen**[36]**, lo cual implica un nuevo conjunto de desafíos que deberán resolverse para poder mantener un framerate aceptable.



12. BIBLIOGRAFÍA

[1] Stam, J., Stable fluids. En *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1999. p. 121-128.

[2] Goodnight, N., Lewin, G., Luebke, D., Skadron, K., A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. *ACM SIGGRAPH 2005 Courses*. ACM, 2005. p. 193.

[3] NVIDIA Optix. https://developer.nvidia.com/optix - Ultima visita: 13/03/2017

[4] Liu, Y., Liu, X., Wu, E., Real-time 3D fluid simulation on GPU with complex obstacles. En *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on.* IEEE, 2004. p. 247-256.

[5] Niemeyer, K., Sung C., Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing*, 2014, vol. 67, no 2, p. 528-564.

[6] Chen, S., Doolen, G., Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics*, 1998, vol. 30, no 1, p. 329-364.

[7] Andersson, J., Karlsson, D. Improving rendering times of Autodesk Maya fluids using the GPU. 2008

[8] Stam, J., Real-time fluid dynamics for games. In *proceedings of the game developer conference*. 2003. p. 25.

[9] Amouzgar R., Liang Q., Clarke PJ., Yasuda T., Mase H.. Computationally efficient tsunami modelling on Graphics Processing Units (GPUs). *International Journal of Offshore and Polar Engineering* 2016, 26(2), 154-160.

[10] Kajiya, J. T., von Herzen, B. P., Ray Tracing Volume Densities. In *ACM Siggraph Computer Graphics*. ACM, 1984. p. 165-174.

[11] Fedkiw, R., Stam, J., Jensen, Henrik Wann. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001. p. 15-22.

[12] Gary D. Yngve, J., F. O'Brien, K. Hodgins, J., Animating Explosions. *In Proceedings of ACM SIGGRAPH 2000*, pages 29–36, August 2000.

[13] M. Cohen, J., Tariq, S., Green, S., Interactive fluid-particle simulation using translating Eulerian grids. *In Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010. p. 15-22.

[14] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, Ken Perlin. Accelerating Eulerian Fluid Simulation With Convolutional Networks. *arXiv preprint arXiv:1607.03597*, 2016.

[15] Harris, M. J., Fast fluid simulation on the GPU. In *SIGGRAPH Courses*. 2005. p. 220.

[16] White, Frank M., Corfield, I., *Viscous fluid flow*. Boston: McGraw-Hill Higher Education, 2006.

[17] Maria Denaro, F. On the application of the Helmholtz–Hodge decomposition in projection methods for incompressible flows with general boundary conditions. *International Journal for Numerical Methods in Fluids*, 2003, vol. 43, no 1, p. 43-69.

[18] Bridson R., Fluid Simulation For Computer Graphics. CRC Press, 2015.

[19] Stam J., The art of fluid animation. A K Peters. 2015.

[20] Chorin, A., A Numerical Method for Solving Incompressible Viscous Flow Problems, *Journal of Computational Physics*, vol. 135, no. 2, Elsevier, 1967.

[21] Marsden, J. E., Tromba, A., *Vector calculus*. Macmillan, 2003.

[22] Foster , N., Metaxas, D., Realistic Animation of Liquids. *Graphical Models and Image Processing. 1996, vol. 58, no 5, p. 471-483.*

[23] Courtecuisse, H., Allard, J., Parallel dense gauss-seidel algorithm on many-core processors. In *High Performance Computing and Communications*. 11th IEEE International Conference on. IEEE, 2009. p. 139-147.

[24] Cieplak, M., Koplik, J., Banavar, J. R., Boundary conditions at a fluid-solid interface. *Physical Review Letters*, 2001, vol. 86, no 5, p. 803.

[25] Fujii, Y., et al. Data transfer matters for GPU computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on. IEEE*, 2013. p. 275-282.

[26] NVIDIA, Cuda shared memory, https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ - última visita: 24/08/2017

[27] OpenGL, Textures, https://open.gl/textures - última visita: 12/06/2017

[28] NVIDIA. Texture and surface memory. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-and-surface-memory - última visita: 12/06/2017

[29] Engel, W., *GPU Pro 7: Advanced Rendering Techniques*. Capítulo 6.2, CRC Press, 2016.

[30] Unity Technologies, Unity3D's rendering pipeline, https://docs.unity3d.com/Manual/SL-RenderPipeline.html - Ultima visita: 03/7/2017

[31] Microsoft, Direct3D 11 Rendering pipeline, https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx - última visita: 03/7/2017

[32] Microsoft, Common shader core, https://msdn.microsoft.com/en-us/library/windows/desktop/bb509580(v=vs.85).aspx - última visita: 03/7/2017

[33] JANßEN, C. F., Rung, T., GPUs and LBM: a perfect match for real-time simulations and interactive monitoring of three-dimensional CFD.

[34] Maccormack, R. W., The effect of viscosity in hypervelocity impact cratering. *Frontiers of Computational Fluid Dynamics,* 1969, p. 27-44.

[35] Bolz, J., et al. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG),* 2003, vol. 22, no 3, p. 917-924.

[36] Bentoumi, H., Gautron, P., Bouatouch, K., GPU-based volume rendering for medical imagery. *Int J Electr Electron Eng*, 2010, vol. 4, no 1.

[37] NVIDIA. Fermi compute architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf - última visita: 5/6/2017

[38] NVIDIA. Life of a triangle. https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline - última visita: 10/3/2017

[39] Elsen, E., Legresley, P., Darve, E., Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 2008, vol. 227, no 24, p. 10148-10161.

[40] Braley, C., Sandu, A., Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. *Virginia Tech, Blacksburg*, 2010.

[41] NVIDIA - Global memory usage and strategy https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf - *GPU Computing Webinar 7/12/2011.* Última visita: 28/8/2017.

[42] De vuyst, F., Labourdette, C., Rey, C., GPU-accelerated real-time visualization and interaction for coupled Fluid Dynamics. *21ème Congrès Français de Mécanique, 26 au 30 août 2013, Bordeaux, France (FR)*, 2013.

[43] Amador, G., Gomes, A., Linear solvers for stable fluids: GPU vs CPU. *17th Encontro Portugues de Computacao Grafica (EPCG09)*, 2009, p. 145-153.

[44] Arce Acuña, Marlon & Aoki, Takayuki. (2009). Real-Time Tsunami Simulation on Multi-node GPU Cluster.

[45] NVIDIA - Advanced CUDA Webinar Memory Optimizations. 2009 http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf .Última visita: 1/10/2017.

[46] Micikevicius, P. GPU performance analysis and optimization. En *GPU technology conference*. 2012.

[47] OpenGL - Matrices modelo, vista y proyección http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/#matrices-modelo-vista-y-proyecci%C3%B3n .Última visita: 3/5/2017.

[48] Microsoft - High Level Shading Language https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx .Ultima visita: 1/2/2017

APÉNDICE A - Glosario

advección: la aplicación de movimiento a alguno de los atributos del fluido, por ejemplo la densidad.
advective rate of change: la velocidad a la cual un atributo del fluido es adveccionado.
alpha blending: es la composición de dos imágenes (superpuestas), una con un grado de transparencia (alpha) el cual determinará la cantidad de color a utilizar de una u otra imagen.
backface culling, front face culling: eliminación de triángulos de la geometría en base a la orientación de la cara de dichos triángulos.
backwards advection: método de advección propuesto por Jos Stam.
bilinear interpolation: doble interpolación lineal entre 4 elementos.
binding semantic: etiqueta utilizada para los atributos de un shader.
blending mode: modo que define la forma en la cual se compondrán las imágenes (por ejemplo alpha blending).
cell-centered grid: grilla que almacena los valores en el centro de cada celda.
CGI: imágenes generadas por computadora.
clipping volume: espacio de coordenadas con un rango de -1..1 en **X** e **Y**, y 0..1 en **Z**.
clip space o projective space: espacio de coordenadas en el que deben estar las posiciones de los vértices antes de ser recibidos por el rasterizer.
common shader core: estructura que define la funcionalidad y acceso a recursos que tendrá cada etapa del rendering pipeline.
compute shader: programa de "propósito general" ejecutado en la GPU independiente del rendering pipeline.
ConvNet - convolutional neural network: red neuronal entrenada para la simulación de fluidos.
curl - la cantidad de rotación que posee un vector en un punto dado.
device - manera de referirse a la GPU.
Eulerian grid: grilla Euleriana en donde se almacenan los valores del fluido en puntos fijos (celdas) del espacio.
fixed stage: tipo de etapa en la cual su funcionalidad está definida y no puede ser modificada.
fluid in a box: tipo de fluido el cual está contenido en una caja, en donde nada puede entrar ni salir.
flux: flujo de velocidad o atributos similares.
frame: imagen.
framerate: imágenes por segundo.
free-slip condition: condición que define el comportamiento que tendrá el fluido en las fronteras, el cual evitará que este escape del entorno.
game engine: conjunto de herramientas, tecnologías, y librerías integradas para el desarrollo de aplicaciones interactivas.
HLSL: High Level Shading Language, lenguaje utilizado para programar los shaders de D3D y DirectCompute.
host: manera de referirse a la CPU.
interactive rendering: renderizado a una velocidad de entre 1 y 24 frames por segundo.

kernel: programa a ser ejecutado en la GPU.
local rate of change: cantidad de cambio de un atributo que se produce en un punto fijo.
multigrid method: método utilizado para resolver ecuaciones diferenciales. Útil para la aceleración de los cálculos durante la simulación.
off-chip/on-chip: ubicación dentro o fuera del chip de la GPU.
off line rendering: renderizado a menos de 1 frame por segundo, comúnmente visto en las aplicaciones utilizadas en la industria cinematográfica para generar imágenes.
Operator splitting: método para resolver una gran ecuación dividiéndola en términos y luego combinando los resultados.
primitive clipping: recorte de primitivas que estén por fuera del clipping volume.
primitive culling: eliminación de primitivas que estén por fuera del clipping volume.
programmable stage: etapa programable del rendering pipeline.
ray tracing: técnica utilizada para el renderizado de imágenes.
real time rendering: generación de imágenes a una velocidad mayor de 24 frames por segundo.
rendering pipeline: conjunto de etapas por los que deben pasar los datos en la GPU para generar una imagen.
renderización: generado de una imagen.
render target: buffer que representa una textura sobre la cual se escribirán los resultados luego de una ejecución del rendering pipeline.
resource binding: asociación de recursos con las etapas del rendering pipeline.
Sparse simulation: técnica utilizada para reducir la memoria consumida en simulaciones de tres dimensiones.
staggered grid: grilla que almacena sus valores en los bordes de cada celda.
steady state: estado de estabilidad al que se llega con el paso del tiempo.
texel: elemento de una textura.
time step: intervalo de tiempo entre cada "paso" de la simulación.
trilinear interpolation: interpolación entre 4 elementos de una textura teniendo en cuenta sus niveles de mip maps.
velocity field: campo vectorial que representa la velocidad del fluido.
vertex skinning: técnica utilizada para animar los vértices de una geometría utilizando un esqueleto compuesto de huesos virtuales.
VFX: efectos especiales generados por computadora.
vorticity confinement: técnica utilizada para inyectarle al fluido la fuerza que se pierde a causa de la disipación.

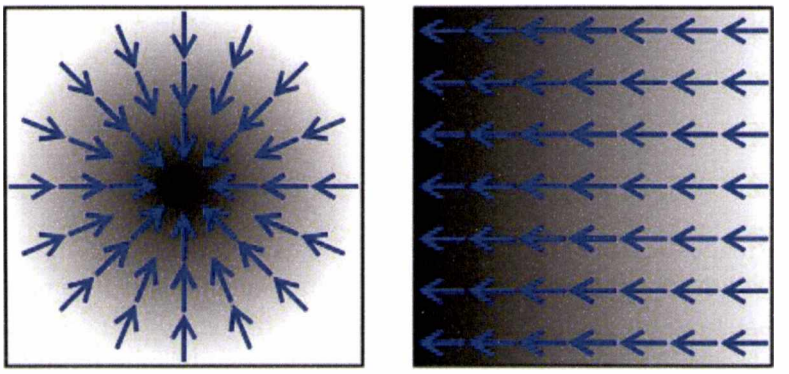
APÉNDICE B - Operadores del cálculo vectorial

En esta sección se repasan los operadores del gradiente, divergencia, curl y laplaciano, los cuales son necesarios para comprender y discretizar las ecuaciones que modelan el comportamiento de los fluidos.

Gradiente

El gradiente representa la dirección en la que se produce el mayor incremento en una función de múltiples variables. Dado que representa una dirección, el resultado del gradiente es un vector.

En la siguiente imagen se ilustra el resultado del gradiente tomado en distintos puntos de una función bidimensional:



El gradiente puede ser definido en dos dimensiones de la siguiente forma:

$$\nabla f(x,y) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

Como se puede observar, consiste en calcular la derivada de la función en cada uno de los ejes de la función. También es posible utilizar la siguiente notación:

$$\nabla f = \frac{\partial f}{\partial \vec{x}}$$

Utilizar un vector en el denominador de la derivada parcial indica que se tomará la derivada con respecto a cada uno de los componentes de ese vector.

Divergencia

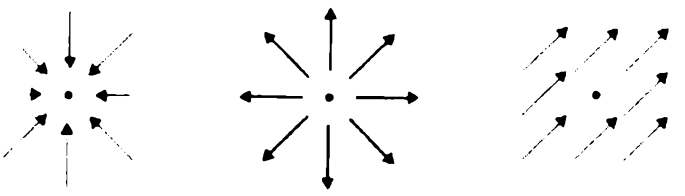
El operador de divergencia solo puede ser aplicado sobre campos vectoriales, y mide qué tanto los vectores convergen o divergen alrededor de un punto. En dos dimensiones se define de la siguiente manera:

$$\nabla \cdot \vec{u} = \nabla \cdot (u,v) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

El cálculo de la divergencia consiste en calcular las derivadas de cada uno de los componentes del vector, cada uno con respecto a los ejes **x** e **y** del campo vectorial, y por último sumar los resultados. Nótese que el operador de divergencia recibe un vector y retorna un valor escalar.

Si la divergencia es negativa, indica que los vectores convergen hacia un punto. Si la divergencia es positiva, los vectores se alejan del punto en cuestión. Si la divergencia es cero, quiere decir que no hay ni convergencia ni divergencia. Esto es ilustrado en la siguiente imagen:

$$\nabla \cdot \vec{\mathbf{v}} < 0 \qquad \nabla \cdot \vec{\mathbf{v}} > 0 \qquad \nabla \cdot \vec{\mathbf{v}} = 0$$

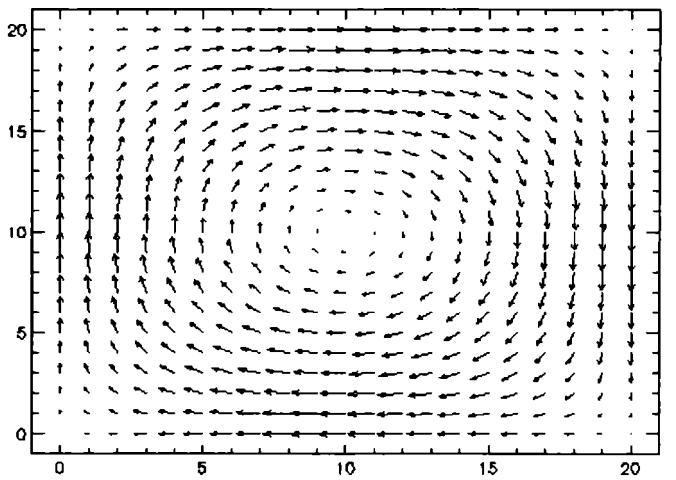


Curl

El operador de curl se aplica sobre un punto perteneciente a un campo vectorial, y se obtiene como resultado un vector que representa la rotación que se encuentra en la región cercana a dicho punto. La dirección de dicho vector indica el eje de rotación, y su magnitud indica la magnitud de la rotación. En dos dimensiones, el operador de curl puede ser definido de la siguiente manera:

$$\nabla \times \vec{u} = \nabla \times (u, v) = \left(\frac{\partial u}{\partial y}, -\frac{\partial v}{\partial x}\right)$$

En la siguiente imagen se ilustra el resultado de aplicar el operador de curl a cada uno de los puntos en el campo vectorial.



Una propiedad del curl que en esta tesina es utilizada para la resolución de la ecuación de Poisson, indica que el curl del gradiente de cualquier campo escalar Φ que sea doblemente diferenciable, da como resultado el vector cero. Es decir:

$$\nabla \times (\nabla \Phi) = 0$$

7.2.4 Laplaciano

El operador laplaciano es utilizado para cuantificar la tasa de cambio que se produce entre un punto y sus vecinos. En dos dimensiones se lo define de la siguiente forma:

$$\nabla \cdot \nabla f = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y}$$

A veces también se suele utilizar la siguiente notación:

$$\nabla^2 f = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y}$$

El operador puede ser aplicado a un campo vectorial o matricial, y el resultado será un número escalar. Cuando la ecuación se la escribe de la siguiente manera, es llamada ecuación de Laplace:

$$\nabla^2 f = 0$$

Y al reemplazar el lado derecho por una constante **b** distinta de 0, la ecuación es llamada ecuación de Poisson:

$$\nabla^2 f = b$$