GPUs: Rigid Body Simulation

Álvaro del Monte Freitas
Dirigido por Pedro Jesús Martín de la Calle
Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid

22 de Junio de 2007

Índice general

Índice de figuras

Capítulo 1

Introducción

En los últimos años la evolución de las tarjetas gráficas ha sido más rápida que la de las CPUs. Esta evolución ha permitido que las GPUs no sólo sean más rápidas y eficientes a la hora de desempeñar sus tareas, sino que puedan desempeñar tareas distintas a las estrictamente gráficas o distintas de las que motivaron su diseño. En efecto, hay ciertas etapas de la tubería gráfica que pueden ser programadas por el programador para sustituir la funcionalidad por defecto de la tarjeta por otro tipo de funcionalidad, que puede no tener que ver con el procesado gráfico. Así ha surgido el concepto de programación genérica en GPUs, denominado GPGPU, y que se basa en aprovechar el potencial de las GPUs para realizar cálculos y de este modo liberar la CPU de cierta carga de trabajo. Además de distribuir el cómputo entre la CPU y la GPU, la principal ventaja consiste en que los cálculos se pueden realizar en GPU muy rápidamente por dos motivos: mayor capacidad de cómputo de la GPU y paralelización del código que se ejecuta. Gracias a este aumento de velocidad se puede conseguir una mayor eficiencia en los programas, aunque no debemos olvidar que no se puede usar la tarjeta gráfica sin antes haber adaptado el código.

El proyecto consistirá en el uso de la GPU en la simulación de sólidos rígidos. Se busca utilizar la tarjeta gráfica para realizar ciertas tareas que de otra forma debería efectuar la CPU. Para ello habrá que estudiar las tareas que son susceptibles de ser paralelizadas y ejecutadas por la GPU. Se han realizado varias versiones del proyecto. Todas ellas son en dos dimensiones, para facilitar los cálculos y su implantación sobre la tarjeta gráfica. En una primera versión se ha estudiado la simulación de sólidos rígidos cuando los objetos son círculos. Con esto se ha pretendido reducir la complejidad de tareas como la colisión entre objetos. Después, se ha desarrollado una versión en la que los objetos son polígonos convexos. Aquí aparecen más tareas (como por ejemplo el uso de aligned bounding boxes) y más complejas. Para poder realizar una comparativa entre CPU y GPU, además de estas dos versiones se han implementado sus correspondientes versiones sobre CPU.

En los siguientes capítulos se verán todos los conceptos implicados que son necesarios

para entender el proyecto. En el capítulo 2 se introducen conceptos básicos relacionados con las GPUs, sus aplicaciones, y se detalla brevemente cómo usar la tarjeta para la programación genérica. En el capítulo 3 se muestra cómo simular el comportamiento del sólido rígido, es decir, se describen todos los conceptos físicos y matemáticos necesarios para poder realizar la simulación, la detección de colisiones y la respuesta a las mismas. En el capítulo 4 se analizará cómo se ha organizado el código y en qué partes interviene la GPU. Por último, en el capítulo 5 se muestra la comparativa de los resultados obtenidos para las implementaciones sobre GPUs y sus correspondientes versiones sobre CPU.

Capítulo 2

GPUs

2.1. Introducción

En los últimos años se han conseguido grandes avances en el campo de las tarjetas gráficas, además la evolución de las GPUs ha sido mucho mayor que la de las CPUs. Estos avances han permitido utilizar la tarjeta gráfica para realizar cálculos que pueden no tener que ver con las tareas de procesado gráfico habituales en ellas. Gracias a esto se puede liberar la CPU de trabajo, consiguiendo así un incremento en el rendimiento de los programas. En los apartados siguientes de este capítulo veremos cómo podemos aplicar estas mejoras.

2.2. Conceptos básicos relacionados con la GPU

Antes de poder programar sobre nuestra tarjeta gráfica tendremos que saber si es posible programar sobre ella y qué partes son programables.

Lo primero será comprobar si la tarjeta es compatible con OpenGL 2.0 o superior, ya que es a partir de esta versión cuando aparecen las extensiones que ofrecen la funcionalidad referente a los shaders.

Después debemos ver qué partes de la GPU son programables. En la sección 2.3 veremos las etapas que actualmente son programables en una tarjeta gráfica: la de procesado de vértices y la de procesado de fragmentos. Nuestra GPU puede que no sea programable para ninguna de estas etapas, que sea programable sólo para una de ellas, o que lo sea para las dos. Para conocer estos datos nos debemos fijar en el tipo y versión de shaders que soporta la GPU en concreto. Cuanto mayor sea la versión de shader, mayor será la funcionalidad soportada por la tarjeta. Actualmente, la versión más moderna de los shaders

es la 4.0.

Uno de los aspectos más importantes es saber el número de unidades de procesado de vértices y fragmentos que tiene la tarjeta. Normalmente las tarjetas disponen de más unidades de procesamiento de fragmentos. La cantidad de unidades se corresponde con el número de vértices o de fragmentos que nuestra tarjeta es capaz de procesar a la vez. Es esta característica la que determina la velocidad de ejecución de nuestros programas, ya que el código se ejecuta de forma paralela usando las distintas unidades de procesado de la tarjeta.

En las siguientes secciones de este capítulo veremos todo lo necesario para poder programar sobre GPU.

2.3. Shaders

Como ya hemos visto las GPUs actuales permiten al programador configurar ciertas etapas del procesado gráfico. Las etapas que forman la tubería gráfica de OpenGL son las siguientes:

- Geometría
- Procesado de vértices
- Ensamblado de primitivas
- Recortado, proyección, puerto de vista
- Rasterizado
- Procesado de fragmentos
- Operaciones por fragmento
- Operaciones de Frame Buffer

Durante años todas las operaciones realizadas por la tarjeta gráfica, y que sucedían en cada etapa de la tubería gráfica, estaban predefinidas y no podían ser modificadas por el programador. Con el paso del tiempo las GPUs han evolucionado llegando a permitir la modificación de algunas de estas etapas. Hoy en día únicamente son programables las etapas de procesado de vértices y de procesado de fragmentos. En un futuro se espera que alguna de las otras etapas pueda ser también configurable.

Una vez que sabemos que las GPUs son programables, hay que ver qué elementos son necesarios en la programación de una tarjeta gráfica. Es en este punto donde aparece el

concepto de *shader*. Un shader se puede definir como el código que se ejecuta en las etapas configurables, con el fin de sustituir la funcionalidad predefinida de dichas etapas. Como hemos visto, actualmente existen dos etapas configurables, por tanto existen dos tipos de shaders:

- Shaders de vértices
- Shaders de fragmentos

Si deseamos usar algún shader es necesario indicar a la tarjeta que queremos usarlo, ya que, si no le indicamos nada, el procesado gráfico se realizará usando la tubería gráfica por defecto. Ambos tipos de shaders pueden usarse conjuntamente o, si lo deseamos, podemos emplear solamente uno de ellos, y mantener para el otro la funcionalidad predefinida. También es posible tener varios shaders del mismo tipo, en este caso tendremos que indicar a la tarjeta qué shader queremos que utilice antes de que comience el procesado gráfico de la etapa correspondiente.

Llegados a este punto una pregunta que nos puede surgir es, si hay dos tipos de shaders, cuál debemos usar. La respuesta varía según el tipo de aplicación en la que los queramos usar. Si lo que nos interesa es usarlo en programación genérica, lo más habitual es utilizar shaders de fragmentos, ya que actualmente este tipo de shaders nos permiten leer y escribir en texturas, y por tanto podemos usar las texturas como entrada y salida de datos. Por el contrario, los shader de vértices únicamente permiten la lectura desde texturas. Otra ventaja de los shaders de fragmentos es que el número de unidades de procesado de fragmentos suele ser superior al número de unidades de procesado de vértices, y por tanto podemos conseguir un mayor rendimiento utilizando los shaders de fragmentos.

En la figura 2.1 se muestra el flujo de datos que tiene lugar a través de los shaders de vértices y de fragmentos. El shader de vértices tiene como entrada atributos de vértices tales como la posición, normal, color, etc. La salida del shader de vértices son valores que serán posteriormente interpolados para formar parte de la entrada del shader de fragmentos. Los valores especiales, referenciados en el diagrama, son valores que deben ser escritos por la tubería gráfica para su correcto funcionamiento. El shader de fragmentos tiene como salida uno o más valores de color, más un posible valor de profundidad. Además, ambos shaders tienen acceso a datos *uniform* y a texturas. Aunque este acceso es únicamente de lectura, las texturas pueden asociarse a buffers de color, permitiendo así que los shaders de fragmentos escriban en ellas.

Para implementar los shaders se pueden utilizar varios lenguajes. Uno de estos lenguajes es GLSL (OpenGL Shading Language). En el siguiente apartado lo describiremos brevemente.

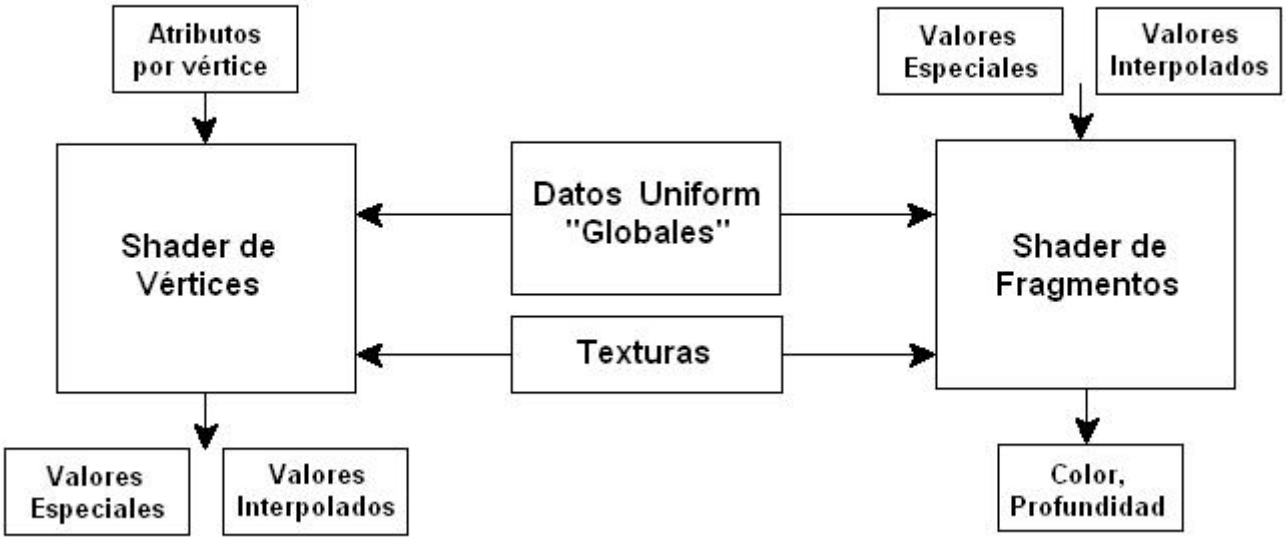


Figura 2.1: Entrada y salida de los shaders

2.4. GLSL

2.4.1. Introducción

OpenGL Shading Language (GLSL) es un lenguaje de programación de shaders de alto nivel basado en el lenguaje de programación C. De hecho, gran parte de la funcionalidad de C ha sido usada en GLSL, aunque también incorpora algunos aspectos de C++. Por esta razón, muchos de los aspectos de GLSL, como las palabras reservadas y la sintaxis, nos resultarán muy familiares por ser parecidos a los usados en otros lenguajes basados en C. A pesar de estas similitudes, también hay muchas diferencias.

Al igual que en C, la ejecución de un programa comienza en la función *main* y termina una vez el programa sale de esta función. Como cualquier otro programa tiene una serie de entradas y de salidas. En el siguiente apartado veremos las principales características de GLSL.

2.4.2. Principales Características de GLSL

A continuación veremos las principales características de este lenguaje de programación. Para más detalle acudir a [Ros04] o [Vil].

- **Visibilidad de las variables:** Existen varios tipos de visibilidad.
  - Visibilidad de Programa: Una variable que existe en un programa es visible en los dos tipos de shader.
  - Visibilidad de Shader: Una variable que tiene este tipo de visibilidad sólo podrá ser vista en un shader de vértices o en un shader de fragmentos.
  - Visibilidad de Función: Las variables con este tipo de visibilidad sólo existen cuando la función se está ejecutando y solamente son visibles dentro de ella.
- **Tipos de datos:** Se ofrecen los siguientes tipos de datos:
  - Escalares: float, int, bool.
  - Vectores: Existen varios tipos de vectores según los datos que contengan.
    - vec2, vec3, vec4: Vector de floats
    - ivec2, ivec3, ivec4: Vector de ints
    - bvec2, bvec3, bvec4: Vector de bools

    El número del final indica el número de componentes del vector.
  - Matrices: mat2, mat3, mat4. Del mismo modo que en los vectores, el número del final indica el tamaño, así mat2 es una matriz de tamaño 2x2. Los elementos de la matriz son de tipo float.
  - Estructuras: Al igual que en C, se pueden usar estructuras. Ejemplo:

    ```
    struct vert{
      vec4 posicion;
      vec3 color;
    }
    ```

    Si declarásemos una variable de tipo `vert` llamada `aux`, podríamos acceder a sus componentes por medio de `aux.posicion` y `aux.color`.
  - Arrays: Se pueden declarar arrays de cualquiera de los tipos enumerados anteriormente. La forma de hacerlo es igual que en C. Ejemplo:

    ```
    mat3 foo[];
    ```

    No es necesario inicializarlos con un tamaño.
  - void: Este tipo es usado para indicar que una función no devuelve ningún valor.
- **Calificadores de tipo:** es posible añadir un calificador de tipo a las variables para modificar su comportamiento. Existen los siguientes calificadores:

  - default: Una variable local que puede ser tanto de escritura como de lectura. También puede ser usada como parámentro de entrada de una función.
  - const: Declaramos que la variable es constante en tiempo de compilación. También indica que la variable puede ser usada como un parámetro de entrada en las funciones y es sólo de lectura.
  - attribute: Indica que la variable es un atributo de vértice. El valor de esta variable es actualizado automáticamente con información correspondiente al vértice. Este es otro método de pasar datos al shader de vértices desde el programa que lo llama (pasar datos de CPU a GPU).
  - uniform: Una variable que no cambia tan a menudo como una variable attribute. Este es otro método de pasar información al shader de vértices o de fragmentos desde el programa que lo llama (pasar datos de CPU a GPU).
  - varying: Una variable usada para pasar información entre el shader de vértices y el de fragmentos.
  - in: Indica que la variable es una variable de entrada en una función, y que sólo puede ser leída por la función.
  - out: Indica que la varable es una variable de salida de la función, y que sólo puede ser escrita por la función.
  - inout: Indica que la variable es tanto de entrada como de salida en una función, y por tanto puede ser leída y escrita dentro de la función.

- **Operadores:** Presenta los mismos operadores y con la misma prioridad que C.
- **Control de Flujo:** Para poder controlar la ejecución de un shader existen una serie de instrucciones muy similares a las de C o C++.

  Podemos hacer bucles haciendo uso de *for*, *while* y *do-while*. Para romper un bucle podemos recurrir a las palabras reservadas *break*, o si deseamos, continuarlo con *continue*.

  También podemos usar saltos condicionales mediante las palabras reservadas *if* e *if-else*.

  Por último, también es posible crear funciones, de la misma forma que se haría en C o C++.
- **Variables Predefinidas:** tanto en el shader de vértices como en el shader de fragmentos existen una serie de variables predefinidas.
- **Funciones Predefinidas:** GLSL ofrece una serie de funciones predefinidas. Estas funciones las podemos dividir en nueve tipos: trigonométricas, exponenciales, comunes, geométricas, funciones que trabajan con matrices, de relación entre vectores, de búsqueda en textura, de procesado de fragmento y de generación de ruido.

2.4.3. La API de GLSL

Siempre que queramos usar un shader en nuestro programa, debemos llevar a cabo una serie de pasos:

1. Crear un objeto *shader*.
2. Cargar el código en el objeto.
3. Compilar el código de este objeto.
4. Repetir los pasos anteriores para el resto de shaders (vértices y fragmentos)
5. Crear un objeto *programa*.
6. Asociar todos los objetos shaders al objeto programa.
7. Pedir a OpenGL que enlace el objeto programa.

Es posible tener varios objetos programas con distintos shaders asociados. De esta forma podremos ir intercambiado el objeto programa activo y realizar en cada momento cálculos distintos.

Para poder utilizar shaders es necesario emplear extensiones de OpenGL que nos permiten crear y gestionar shaders [Ast04]. Para facilitar su uso es posible utilizar la librería GLee con el objeto de cargar dinámicamente aquellas extensiones que soporte la tarjeta. Hay que tener en cuenta que, aunque se esté programando con el API de GLSL, es posible que la tarjeta no soporte ciertas llamadas, o ciertos formatos. Las comprobaciones sobre la capacidad de la tarjeta se tienen que hacer en tiempo de ejecución.

En el capítulo 4 veremos algunos aspectos de la implementación del proyecto. Se presentarán las clases GLSLShader y GLSLProgram. Estas dos clases encapsulan el uso de las extensiones, referentes a los objetos shader y los objetos programa, y de esta forma facilitan su uso.

2.5. Programación genérica de GPUs

2.5.1. Introducción

Como ya se ha comentado, las GPUs actuales permiten configurar ciertas etapas del procesado gráfico para realizar en ellas cálculos que no se parezcan a las tareas de procesado gráfico habituales. Nuestro propósito es utilizar esta capacidad de procesado y liberar de carga de trabajo a la CPU, consiguiendo así un incremento de velocidad en nuestros

programas. A la hora de usar la GPU para programación genérica es necesario conocer y tener en cuenta ciertos aspectos y técnicas que nos facilitarán el empleo de la tarjeta para este fin. En el resto de la sección se mostrarán las ideas principales.

2.5.2. Entrada/Salida de datos

Si queremos usar la GPU para programación genérica es necesario establecer un medio de comunicación entre la GPU y la CPU. Existen varias alternativas para conseguir este intercambio de datos, algunas de ellas son las siguientes:

- Pasar datos de CPU a GPU codificándolos en información de vértices (válido únicamente para shaders de vértices).
- Pasar datos de CPU a GPU a través de variables de tipo uniform.
- Uso de texturas para el flujo de datos propio de los shaders de fragmentos.

De todas las posibilidades, la mejor opción es el uso de texturas. Esto es así porque, como ya vimos, tanto el shader de vértices como el shader de fragmentos tienen acceso de lectura a las texturas de la tarjeta. Además el shader de fragmentos también tiene la posibilidad de modificar el valor de las texturas, siempre que la textura haya sido asociada a un buffer de color, sobre el que escribirá. Por otra parte las texturas pueden verse como arrays, en las que varias componentes o *texels* pueden tratarse al mismo tiempo ya que se procesan en paralelo. El número de texels que serán procesados al mismo tiempo dependerá del número de unidades de fragmentos que tenga la tarjeta.

Como acabamos de decir las texturas pueden verse como arrays, es decir, cada texel de la textura se corresponde a una componente del array. La forma de acceder a un texel es a través de las coordenadas de texturas, que es el equivalente a los índices en un array. Según la dimensión de la textura necesitaremos un número distinto de coordenadas. En el caso de texturas de tipo 1D únicamente se necesita una coordenada, para texturas 2D se precisan dos coordenadas y para texturas 3D necesitaremos 3.

El tamaño de las texturas es limitado. Esta limitación es por motivos hardware y por tanto depende exclusivamente de la tarjeta. En general esto no será ningún problema porque el tamaño suele ser muy elevado. Si queremos comprobar el máximo tamaño soportado por nuestra tarjeta podemos obtenerlo usando la siguiente instrucción:

```
int maxtexsize;
glGetIntegerv(GL_MAX_TEXTURE_SIZE,&maxtexsize);
```

Al llamar a la función, la variable `maxtexsize` se actualizará con el valor del tamaño máximo de textura soportado por nuestra GPU.

Otra característica importante de las texturas es que existen varios tipos de texturas y que las texturas tienen una serie de propiedades que pueden variar: *formato de la textura y formato interno.* Dos son los tipos de texturas existentes: *GL_TEXTURE_2D* y *GL_TEXTURE_RECTANGLE_ARB*. Las diferencias entre uno y otro tipo son las siguientes:

- En *GL_TEXTURE_2D* se normalizan las coordenadas de textura en un rango [0,1] × [0,1], independientemente del tamaño [0,M] × [0,N] que tenga la textura. En el caso de *GL_TEXTURE_RECTANGLE_ARB* las coordenadas de textura no son normalizadas. Por este motivo el uso de estas últimas es más intuitivo para el programador.
- En *GL_TEXTURE_2D* el tamaño que se debe usar es potencia de dos, lo que no es necesario si se utiliza *GL_TEXTURE_RECTANGLE_ARB*.

El formato de la textura indica el número de datos por texel. Uno de los formatos más conocidos es el RGBA; en este caso un texel almacena cuatro valores, uno para la componente roja, otro para la azul, otro para la verde, y por último, uno más para la componente alfa. Además de este formato existen muchos más: *GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE* y *GL_LUMINANCE_ALPHA*. En el proyecto usamos solamente los formatos *GL_RGBA* y *GL_LUMINANCE*. El primero lleva cuatro valores por texel, mientras que el segundo sólo lleva uno.

La última propiedad de la textura es el formato interno. Este formato es dependiente de la tarjeta, por ello es específico de la GPU que usemos. En el proyecto se ha utilizado una tarjeta NVIDIA, por lo que los formatos internos posibles son los siguientes: GL_FLOAT_R32_NV, GL_FLOAT_RGB32_NV y GL_FLOAT_RGBA32_NV.

Una vez vistos los distintos tipos de texturas y formatos, vamos a ver cómo crear una textura:

```
//Identificador de la textura
GLuint texID;
glGenTextures (1, &texID);
//Elegir tipo de textura: GL_TEXTURE_2D o GL_TEXTURE_RECTANGLE_ARB)
glBindTexture(texture_target,texID);
//Desactivar filtros y seleccionar wrap mode
//(obligatorio para texturas float)
glTexParameteri(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```
glTexParameteri(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
//Reservar memoria en la tarjeta para la textura
glTexImage2D(texture_target, 0, internal_format,
             texSize, texSize, 0, texture_format, GL_FLOAT, 0);
```

Donde:

texture_target es el tipo de textura.

internal_format es el formato interno, en nuestro caso *GL_RGBA* y *GL_LUMINANCE*.

texture_format es el formato de la textura, *GL_RGBA* y *GL_LUMINANCE* en nuestro caso.

texSize indica el tamaño de la textura.

Una vez creada la textura nos interesará saber escribir en ella. La escritura podrá realizarse desde CPU para pasar datos a GPU, o desde GPU para pasar datos a CPU.

Escritura/Lectura desde CPU a una textura

La lectura y escritura de CPU en una textura es muy sencilla. A continuación se muestra cómo realizar la escritura desde CPU:

```
//Elegir tipo de textura: GL_TEXTURE_2D o GL_TEXTURE_RECTANGLE_ARB
glBindTexture(texture_target, texID);
//Cargar en la textura el contenido del array data
glTexSubImage2D(texture_target,0,0,0,texAncho,texAlto,
 texture_format,GL_FLOAT,data);
```

Donde:

texture_target es el tipo de textura.

texture_format es el formato de la textura, en nuestro caso *GL_RGBA* y *GL_LUMINANCE*.

texAncho y **texAlto** se corresponden con el alto y ancho de la textura.

data es un array con los datos que deseamos cargar en la textura.

Del mismo modo que nos interesa ser capaces de escribir en una textura desde CPU, nos interesará poder leer la textura desde CPU. Para extraer los datos de la textura hasta la CPU utilizaremos las siguientes instrucciones:

```
//Elegir tipo de textura: GL_TEXTURE_2D o GL_TEXTURE_RECTANGLE_ARB
glBindTexture(texture_target,texID);
//Extraer en el array data el contenido de la textura
glGetTexImage(texture_target,0,texture_format,GL_FLOAT,data);
```

Donde:

texture_target es el tipo de textura.

texture_format es el formato de la textura, *GL_RGBA* y *GL_LUMINANCE* en nuestro caso.

data es un array donde queremos almacenar los datos extraidos de la textura.

Escritura/Lectura desde GPU a una textura

En primer lugar vamos a ver cómo escribir en una textura dentro de un shader. Como ya dijimos, los shaders de fragmentos escribían sus resultados en los buffers. Por tanto tendremos que hacerle saber de alguna forma, a la tarjeta gráfica, que queremos escribir en una textura, ya que de lo contrario la tarjeta volcará los datos en el frame buffer para que sean representados en pantalla. En nuestro caso queremos, por el contrario, que los datos de salida sean almacenados en texturas. Para ello tendremos que usar *FBOS* (frame buffers objects) y asociarles las texturas de salida. Vamos a ver cómo crear un FBO:

```
// Identificador del FBO
Gluint fb;
glGenFramebuffersEXT(1, &fb);
// Asociar el FBO para que sea usado en lugar del framebuffer por defecto
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
```

Una vez creado y ligado el FBO, tenemos que asociarle las texturas en las que deseamos que la GPU escriba. Pueden seleccionarse una o varias texturas de salida. Este número tendrá un limite que variará según el modelo de tarjeta gráfica. Para asociar una textura a un FBO usaremos la siguiente instrucción:

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENTi_EXT,
 texture_target, texID, 0);
```

Donde:

GL_COLOR_ATTACHMENTi_EXT: sirve para precisar en qué posición del FBO estamos enlazando la textura, ya que un mismo FBO puede tener varias. Para ello basta con modificar i. Por ejemplo, usaremos GL_COLOR_ATTACHMENT0_EXT y

GL_COLOR_ATTACHMENT1_EXT, cuando tengamos 2 texturas. De esta forma dentro del shader la instrucción `gl_FragData[i]` sirve para escribir en la textura asociada a GL_COLOR_ATTACHMENTi_EXT.

texture_target es el tipo de textura.

textID es el identificador de la textura que queremos asociar al FBO.

La lectura de una textura en GPU se realiza dentro del código del shader. Para realizar esta operación, GLSL nos ofrece una serie de funciones predefinidas en las que, indicando la textura y las coordenadas deseadas, podemos extraer el dato correspondiente a dichas coordenadas. Si se quiere saber más sobre este tema, se pueden consultar las funciones de búsqueda en texturas en algún manual o libro de GLSL [Ast05].

2.5.3. Procesado de Datos

El procesado de datos se lleva a cabo dentro de los shaders que actúan cuando se dibuja algo en la tarjeta. Por tanto el procesado de datos será equivalente a dibujar. La idea es que cada fragmento corresponda a un dato (es decir, a un texel) y por tanto, cuando se procese un fragmento en realidad se estará procesando un dato. Para que todo funcione correctamente es necesario que la correspondencia sea 1 a 1, es decir, un texel por fragmento. Para conseguirla conviene modificar el puerto de vista de tal forma que la pantalla tenga la misma resolución que el tamaño de la textura. Esta configuración ha de realizarse siempre antes de que se produzca el flujo de datos. Las instrucciones empleadas para dicha configuración son las siguientes, donde texSize indica el tamaño de la textura de salida:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, texSize, texSize);
```

Puede darse el caso de que las texturas de entrada sean de distinto tamaño que las texturas de salida, en ese caso se debe utilizar el tamaño de las texturas de salida para configurar la pantalla.

Después de configurar la pantalla será necesario preparar las texturas de entrada y las texturas de salida. Para preparar las texturas de entrada utilizaremos las siguientes intrucciones:

```
//Activar la textura de entrada
```

```
glActiveTexture(GL_TEXTUREi);
//Enlazar a la textura de entrada, la textura con identificador TexID
glBindTexture(texture_target,TexID);
```

Si hubiese varias texturas de entrada, se ejecutaría varias veces este código, modificando la i de GL_TEXTUREi y asociándole la textura que le correspondiese. Para preparar las texturas de salida lo haremos como ya vimos en la sección 2.5.2:

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENTi_EXT,
 texture_target, texID, 0);
```

Después de configurar la pantalla y preparar las texturas de entrada y de salida, tendremos que indicar a la tarjeta que queremos que dibuje en el buffer que tiene asociadas las texturas de salida. Esto lo conseguimos utilizando las intrucciones *glDrawBuffer* o *glDrawBuffers*. Usaremos la primera si sólo tenemos una textura de salida, y la segunda cuando tengamos varias.

Si tenemos una textura de salida:

```
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
```

Si tenemos dos texturas de salida:

```
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT,
 GL_COLOR_ATTACHMENT1_EXT};
glDrawBuffers(2,attachmentpoints);
```

Una vez realizados todos los pasos anteriores (configurar puerto de vista, preparar texturas de entrada y de salida, indicar en qué buffer queremos dibujar) podemos lanzar el flujo de datos. Para ello dibujaremos un rectángulo que tenga el tamaño de la textura. Hemos de recordar que si utilizamos *GL_TEXTURE_2D* las coordenadas de textura hay que escribirlas normalizadas, en otro caso no. Las instrucciones para lanzar el flujo de datos usando *GL_TEXTURE_2D* son:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
  glTexCoord2f(1.0, 0.0); glVertex2f(texSize, 0.0);
  glTexCoord2f(1.0, 1.0); glVertex2f(texSize, texSize);
  glTexCoord2f(0.0, 1.0); glVertex2f(0.0, texSize);
glEnd();
```

Las instrucciones para lanzar el flujo de datos usando *GL_TEXTURE_RECTANGLE_ARB* son:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
  glTexCoord2f(texSize, 0.0); glVertex2f(texSize, 0.0);
  glTexCoord2f(texSize, texSize); glVertex2f(texSize, texSize);
  glTexCoord2f(0.0, texSize); glVertex2f(0.0, texSize);
glEnd();
```

2.5.4. Técnica Ping-Pong

En algunas ocasiones desearemos leer y escribir en una misma textura. Esto no es posible debido principalmente al problema que surge en situaciones como la siguiente: un fragmento intenta acceder a la información de otro fragmento que ha sido procesado anteriormente y, por tanto, modificado. La técnica ping-pong sirve para evitar esta circunstancia, utilizando dos texturas, en lugar de una. De esta forma, una textura se comportará como entrada del shader y la otra como salida. El papel de entrada y salida se irá intercambiando entre las dos texturas si queremos iterar la ejecución del shader.

A continuación se muestra un ejemplo:

```
// Se almacenan los identificadores de las dos texturas tex0 y tex1
GLuint texID[2];
texID[0]=tex0.getId();
texID[1]=tex1.getId();
// En la técnica ping-pong manejaremos estas dos variables para indicar
//cuál es la textura de entrada, y cuál es la de salida. En este caso estamos
//inicializándolo para que texID[0] sea la textura de entrada y texID[1]
//sea la de salida.
int texIn = 0;
int texOut = 1;
.
.
.
//El método pasada prepara todo para que la ejecución del shader sea
//correcta y realice una pasada por el shader
void pasada(){
  //Configurar el puerto de vista para tener una relación 1:1
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, texSize, 0.0, texSize);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
```

```
  glViewport(0, 0, texSize, texSize);

  //Activar la textura de entrada
  glActiveTexture(GL_TEXTURE0);
  //Enlazar la textura de entrada
  glBindTexture(texture_target,texID[texIn]);

  // Asociar la textura de salida al FBO
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                            GL_COLOR_ATTACHMENT0_EXT0,
                            texture_Target, texID[texOut], 0);

  // Seleccionamos en qué buffer queremos que escriba
  glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT0);

  //Flujo de Procesado
  glBegin(GL_QUADS);
    .
    .
    .
  glEnd();

  //Intercambiar la textura de entrada por la de salida y viceversa.
  intercambiar();
}

void intercambiar(){
  if(texIn==0){
    texIn=1; texOut=0;
  }
  else{
    texIn=0; texOut=1;
  }
}
```

2.5.5. Método de minimización

Las operaciones de minimización consisten en reducir un conjunto de datos a un único valor. Un ejemplo donde se aplica este tipo de operación es el cálculo del mínimo de una

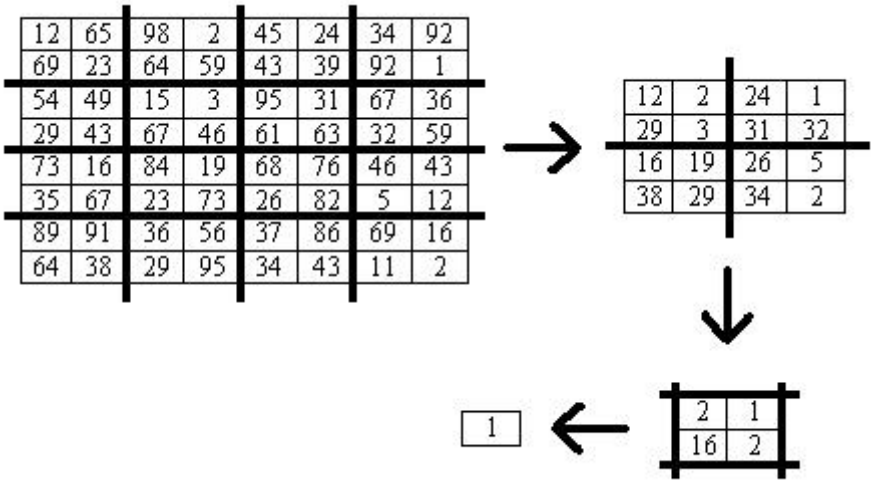


Figura 2.2: Cálculo del mínimo de una matriz

matriz. Para realizar estas operaciones podemos dividir el conjunto inicial en varios subconjuntos, procesar estos subconjuntos en paralelo y de esa forma acelerar el cálculo global. En la Figura 2.2 podemos ver un ejemplo de esta técnica, para calcular el mínimo de una matriz.

Estas operaciones se ajustan bien al procesado mediante shaders, ya que se basan en ejecución en paralelo. La matriz de tamaño 2n $\times$ 2n corresponde a una textura que se framenta en trozos cuadrados de 2 $\times$ 2 texels. Se va reduciendo la matriz, fragmentándola en trozos. Cada uno de esos trozos será procesado y se obtendrá un único valor. Estos valores forman una matriz de tamaño n $\times$ n. Por lo tanto, en cada paso del proceso de minimización, dividimos por 2 cada dimensión de la textura actual. Se van repitiendo estos pasos hasta que se consigue obtener, como salida, un único valor. Para que el método funcione, se requiere trabajar con matrices cuadradas cuyo lado sea potencia de 2. El coste de es logarítmico con respecto al tamaño del lado de la matriz.

Capítulo 3

Simulación del Sólido Rígido

3.1. Introducción

En física se utiliza el concepto de sólido rígido como idealización de los cuerpos sólidos. En un sólido rígido, la distancia entre dos puntos cualesquiera del mismo se mantiene constante, independientemente de las fuerzas que actúen sobre él; es decir, no admite deformaciones. La rama de la física encargada del estudio de este tipo de cuerpos se conoce como *Dinámica del Sólido Rígido*.

3.2. Dinámica del Sólido Rígido

3.2.1. Cinemática del Sólido Rígido

Para poder llevar a cabo la simulación de un objeto debemos almacenar su posición y su orientación. La primera la expresaremos mediante un vector $\mathbf{x}(t)$, que indica la translación del centro de masas del objeto con respecto al origen de coordenadas de la escena. En nuestro caso, la simulación va a ser en 2D, por tanto para representar la posición bastará con un vector con dos componentes (x,y). En cuanto a la forma de representar la rotación, se pueden utilizar varias aproximaciones, pero por tratarse de una simulación 2D se ha considerado que la más adecuada es el ángulo de giro con respecto a su sistema de coordenadas local. La rotación la denominaremos $\mathbf{r}(t)$. Podemos ver un ejemplo de rotación en la Figura 3.1.

La forma geométrica de un sólido rígido puede definirse de forma invariable en un *sistema de coordenadas local*. Si tenemos la descripción geométrica de un objeto en dicho sistema local, podemos transformarla al sistema de coordenadas *global* con ayuda de $\mathbf{x}(t)$

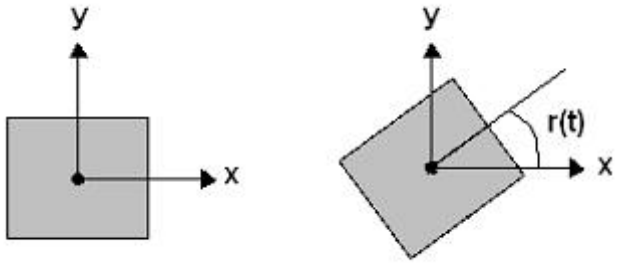


Figura 3.1: Rotación de un objeto

y $\mathbf{r}(t)$. En efecto, las coordenadas de un punto en el sistema de coordenadas local ($\mathbf{p}$) y en el sistema de coordenadas global ($\mathbf{p'}$) quedan relacionadas mediante las siguientes expresiones:

$$\mathbf{p'}.x = \mathbf{p}.x\ \mathbf{cos}(\mathbf{r}(t)) - \mathbf{p}.y\ \mathbf{sin}(\mathbf{r}(t)) + \mathbf{x}(t).x \tag{3.1}$$

$$\mathbf{p'}.y = \mathbf{p}.x\ \mathbf{cos}(\mathbf{r}(t)) + \mathbf{p}.y\ \mathbf{sin}(\mathbf{r}(t)) + \mathbf{x}(t).y \tag{3.2}$$

En donde p.x hace referencia a la coordenada x del vector p. Del mismo modo p.y hace referencia a la coordenada y.

Para modelar el movimiento y el giro de los objetos necesitamos otras dos magnitudes:

- La *velocidad lineal* se define como:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{3.3}$$

- La *velocidad angular*, que en el caso 2D podemos representarla como un único valor que expresa el ángulo de giro por unidad de tiempo:

$$\omega = \frac{d\mathbf{r}}{dt} \tag{3.4}$$

En ciertas ocasiones será necesario calcular la velocidad en un determinado punto $\mathbf{p}$ del sólido rígido en movimiento. La velocidad de dicho punto será igual a la combinación de la velocidad lineal y de la velocidad angular en dicho punto. En la siguiente ecuación podemos ver dicho cálculo:

$$\mathbf{v}_p = \mathbf{v} + \omega \times \mathbf{r}_p \tag{3.5}$$

Donde:

$\mathbf{v}$ es la velocidad lineal del objeto y dicha velocidad se conserva en todos los puntos.

ω es la velocidad angular del objeto.

$\mathbf{r}_p$ es el vector distancia entre el centro de masas del cuerpo y el punto p, es decir:

$$\mathbf{r}_p = p - \mathbf{x}(t)$$

La expresión $\omega \times \mathbf{r}_p$ es un producto vectorial en el que entendemos que ω es, en realidad, el vector (0,0,ω):

$$\omega \times \mathbf{r}_p = (0, 0, \omega) \times (\mathbf{r}_p.x, \mathbf{r}_p.y, 0) = (-\omega \mathbf{r}_p.y, \omega \mathbf{r}_p.x, 0)$$

Al tratarse de dos dimensiones la coordenada z tiene valor cero y, por tanto, sólo nos interesa el vector de dos componentes $(-\omega \mathbf{r}_p.y, \omega \mathbf{r}_p.x)$.

3.2.2. Propiedades de masa del Sólido Rígido

Las propiedades de masa de un sólido rígido son importantes en el estudio del comportamiento de este tipo de cuerpos, ya que la velocidad (tanto lineal como angular) y la respuesta a la aplicación de una fuerza están en función de dichas propiedades.

En primer lugar, la *masa* de un objeto es la medida que indica la resistencia que éste ofrece al ser desplazado. Cuanto mayor es la masa, más difícil será cambiar su estado de movimiento. La masa total de un cuerpo podría calcularse sumando las masas de todas las partículas elementales que lo conforman; para cada partícula elemental se calculará su masa multiplicando su densidad (ρ) por su volumen. Tendríamos pues una expresión de la forma:

$$m = \int \rho \, dV$$

En el caso de cuerpos con densidad uniforme, la expresión sería $m = \rho V$. No obstante, en este trabajo **la masa de un cuerpo se supondrá ya especificada por el programador**, por lo que no se procederá a su cálculo en ningún caso.

Otro concepto importante es el *centro de gravedad* de un sólido rígido, que es el punto, dentro del mismo, alrededor del cual se encuentra la masa uniformemente distribuida. En términos de dinámica de rotación, es el punto en el que puede actuar cualquier fuerza sin causar una rotación en el sólido. Este punto lo especificamos en el sistema de coordenadas global de la escena; para su cálculo habría que dividir el sólido en infinitas masas elementales, cada una con su centro, para posteriormente multiplicar el centro de cada una y su masa, sumarlas todas, y dividir el resultado entre la masa total del sólido. Esto conllevaría el cálculo de integrales sobre el volumen del cuerpo, al igual que en la ecuación anterior.

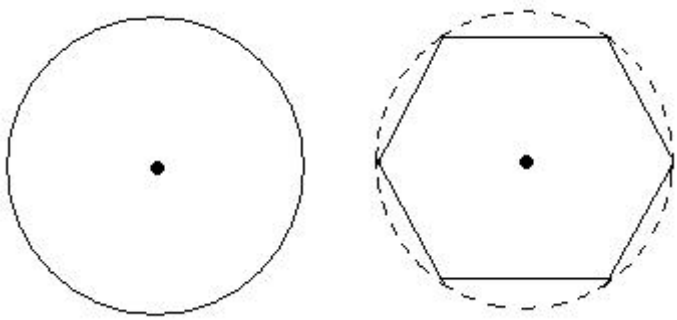


Figura 3.2: Posición del centro de masas para un círculo y un hexágono regular

No obstante, en nuestro caso podemos dividir el sólido en una cantidad finita n de masas elementales, obteniendo la siguiente expresión:

$$\mathbf{CM} = \frac{(\sum_{i=1}^{n} x_i m_i \ , \sum_{i=1}^{n} y_i m_i \ , \sum_{i=1}^{n} z_i m_i)}{\sum_{i=1}^{n} m_i} \tag{3.6}$$

Los objetos que nosotros vamos a utilizar son **círculos y polígonos regulares**, por tanto el cálculo del centro de masas es muy sencillo de calcular y no requiere utilizar la ecuación 3.6. En el caso de un círculo, el centro de masas se correspondonde con el centro del mismo. Para los polígonos regulares este punto se encuentra en el centro de la circunferencia que lo circunscribe. En la Figura 3.2 podemos ver el centro de masas en el caso de un círculo y de un hexágono regular.

Otra propiedad de relevancia para el movimiento de los cuerpos es su *momento de inercia*. El momento de inercia mide la distribución radial de la masa de un cuerpo a lo largo de un eje de rotación. Al igual que la masa indicaba la resistencia de un objeto al movimiento *lineal*, el momento de inercia indica la resistencia de un objeto al movimiento *rotacional*, pero a diferencia de la anterior, este varía según el eje de rotación que consideremos.

El valor del momento de inercia depende del eje de rotación que se considere, aunque, en el caso 2D, el eje de rotación será perpendicular al plano, por lo que puede expresarse con un simple escalar. Al momento de inercia lo denotaremos **I**.

I es un valor constante a lo largo del tiempo y es calculado al principio, en función de la geometría del cuerpo. A continuación veremos cómo calcular el momento de inercia para polígonos regulares. En el caso de los círculos no vamos a tener en cuenta la rotación, porque no se apreciaría, y en consecuencia no será necesario tampoco tener en cuenta los momentos de inercia.

- **Momento de Inercia de un Polígono Regular:** Se quiere calcular el momento de inercia en el que el eje de giro pasa por el centro del polígono (Eje z). Para calcularlo,

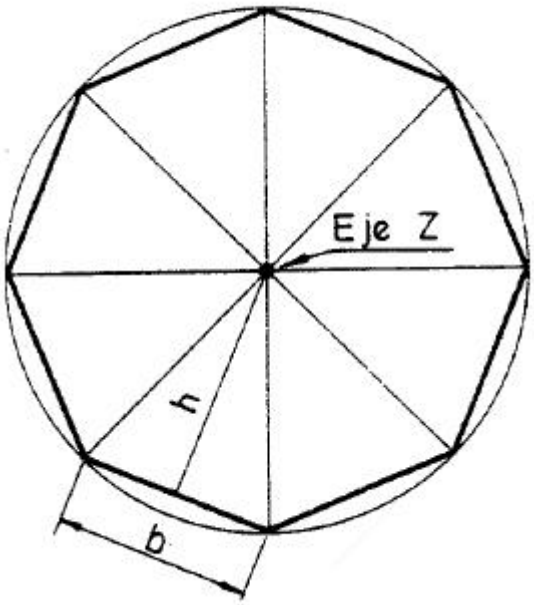


Figura 3.3: División de un polígono regular en triángulos isósceles

se divide el polígono en tantos triángulos isósceles como lados n tiene el polígono [Asc], como se ve en la Figura 3.3. Por lo tanto, el momento de inercia del polígono será n veces el momento de inercia de uno de los triángulos isósceles cuyo eje pase por el vértice. El momento de inercia de un triángulo isósceles, en estas condiciones, podemos calcularlo mediante la siguiente fórmula:

$$I_{isosceles} = \frac{masa_{isosceles}}{24}(b^2 + 12 \cdot h^2)$$

donde b y h son la base y la altura del triángulo, respectivamente.

El momento de inercia del polígono será, por tanto:

$$I_{poligono} = n \cdot I_{isosceles} =$$
$$= n(\frac{masa_{isosceles}}{24}(b^2 + 12 \cdot h^2)) =$$
$$= (\frac{masa_{poligono}}{24}(b^2 + 12 \cdot h^2))$$

Observar que $n \cdot masa_{isosceles}$ es igual a la masa total del polígono, ya que dicho polígono está formado por n triángulos isósceles iguales.

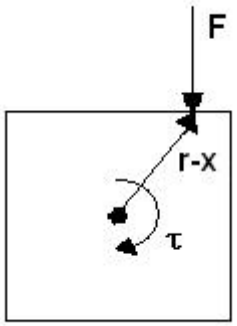


Figura 3.4: Rotación provocada por una fuerza

3.2.3. Dinámica del Sólido Rígido

Los responsables de causar cambios en la velocidad de un objeto son las *fuerzas*. Según la Primera Ley de Newton, todo cuerpo persevera en su estado de reposo o movimiento a menos que la acción de una fuerza le obligue a cambiarlo.

En el estudio de un sólido rígido se realiza distinción entre *fuerza* ($\mathbf{F}$) y *torque* (τ). La primera es la responsable del movimiento lineal de una partícula, mientras que el segundo es el responsable del movimiento rotacional y se expresa en términos de $\mathbf{F}$. El torque se define como:

$$\tau = (\mathbf{r} - \mathbf{x}) \times \mathbf{F} \tag{3.7}$$

Donde $\mathbf{r}$ es el punto de aplicación de la fuerza (en el sistema de coordenadas global) y $\mathbf{x}$ es la posición del centro de masas del objeto. Por tanto, cuanto mayor es la distancia del punto de aplicación al centro de masas, mayor será el torque y viceversa. Además, la dirección del torque es la misma que la dirección de la velocidad angular que este provoca (Figura 3.4).

Cuando en un cuerpo actúan varias fuerzas $\mathbf{F}_i$, la fuerza externa total $\mathbf{F}$ es la suma de todas ellas. Análogamente, el torque externo total es la suma de todos los torques.

El *momento lineal* de una partícula se define como el producto de la masa por su velocidad ($\mathbf{p} = m\mathbf{v}$). En el caso de un sistema de partículas, el momento total del sistema es igual a la suma del momento de todas las partículas que lo forman:

$$\mathbf{P} = \sum_{i=1}^{n} \mathbf{P}_i$$

Un resultado importante es que el momento lineal total de un sólido rígido es el mismo que el momento lineal que tendría una partícula situada en su centro de gravedad con la

misma masa M y velocidad v que el sólido:

$$\mathbf{P} = M\mathbf{v} \tag{3.8}$$

Ahora bien, necesitamos una ecuación que relacione la fuerza total externa aplicada sobre un objeto y el cambio de velocidad lineal que éste sufre como consecuencia. Esta relación viene dada por la *Segunda Ley de Newton*:

$$\boxed{\mathbf{F} = \frac{d\mathbf{P}}{dt}} \tag{3.9}$$

Para describir el cambio de velocidad angular de un objeto necesitamos una magnitud análoga al momento lineal. Esta magnitud es el *momento angular* (**L**), que viene definido por la ecuación:

$$\mathbf{L} = \mathbf{I}\omega \tag{3.10}$$

Y del mismo modo que la fuerza total provoca variación en el momento lineal, es el torque quien provoca variación en el momento angular:

$$\boxed{\tau = \frac{d\mathbf{L}}{dt}} \tag{3.11}$$

En este trabajo, las **fuerzas** que actúan en los objetos no pueden ser introducidas por el usuario, sino que **son el resultado de las colisiones que tienen lugar entre los objetos**. Veremos cómo calcular dichas fuerzas en el apartado 3.5.

3.2.4. Simulación de Sólidos Rígidos independientes

En este trabajo la simulación se realiza por *ciclos*. Existe un estado para cada objeto, que indica su posición, orientación, etc. En cada paso de simulación se comprueba si existen colisiones a lo largo del ciclo. Para los objetos que colisionan se calculan las fuerzas resultantes, y los torques correspondientes. Posteriormente se actualiza el estado de los objetos atendiendo a estas fuerzas.

Las magnitudes que se almacenan para cada objeto (y, por tanto, definen su estado) son:

- Posición
- Orientación
- Velocidad lineal

- Velocidad angular

Estas cuatro magnitudes conforman el *vector de estado* $\mathbf{Y}$:

$$\mathbf{Y}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{r}(t) \\ \mathbf{v}(t) \\ \omega(t) \end{pmatrix} \tag{3.12}$$

Ahora bien, para avanzar un ciclo de la simulación necesitaremos saber la *variación* de dicho estado:

$$\frac{d\mathbf{Y}(t)}{dt} = \begin{pmatrix} d\mathbf{x}(t)/dt \\ d\mathbf{r}(t)/dt \\ d\mathbf{v}(t)/dt \\ d\omega(t)/dt \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \omega(t) \\ \frac{\mathbf{F}(t)}{M} \\ \frac{\tau(t)}{I} \end{pmatrix} \tag{3.13}$$

¿Podemos conocer la variación a partir de nuestra representación del estado? La respuesta es sí, ya que:

- La fuerza total ($\mathbf{F}$) la podemos conocer sumando las fuerzas individuales que actúan sobre nuestro objeto, y éstas son calculadas según se explica en la sección 3.5.
- El torque (τ) se puede calcular mediante su definición a partir de la ecuación (3.7).

El cálculo del nuevo estado $\mathbf{Y}'$ a partir del estado inicial $\mathbf{Y}$ y de d$\mathbf{Y}$/dt requiere la utilización de métodos numéricos para el cálculo de ecuaciones diferenciales, que se verán en la sección 3.3.

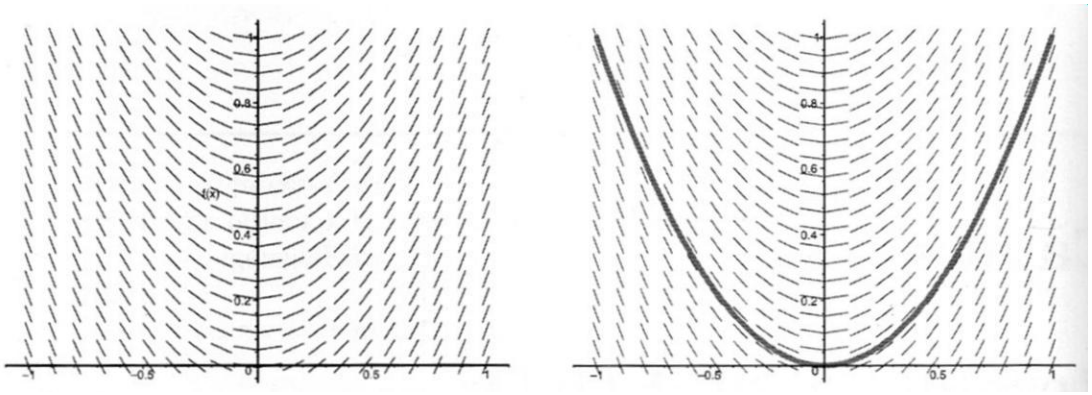
3.3. Resolución de EDOs

Hemos visto anteriormente que la simulación se realiza por pasos. Cada paso consistía en la variación del estado de los objetos de la escena. Dicha variación venía representada por la ecuación 3.13.

Simplificando el problema al caso de una única dimensión, tratamos de resolver ecuaciones del tipo:

$$\frac{du}{dt} = f(u, t)$$

Esta es la que se conoce como *ecuación diferencial ordinaria* (EDO). Gráficamente podemos visualizarla como un campo vectorial (Figura 3.5(a)), donde se indica la variación de



(a) Sin valor inicial (b) Con valor inicial (0,0)

Figura 3.5: Campo vectorial originado por la EDO $du/dt = 2t$

u. Particularmente estamos interesados en asociar un *valor inicial* a la ecuación diferencial, de modo que la función que obtengamos como solución de dicha ecuación sea única (Figura 3.5(b)).

Esta clase de problemas son especialmente útiles en la simulación de sólidos rígidos, donde nuestros objetos tienen un **estado inicial**. Partiendo de dicho estado (ecuación 3.12), y mediante las cuatro ecuaciones que determinan su variación (ecuación 3.13), podemos calcular la evolución de un objeto a lo largo del tiempo.

Dado que la variación del estado de un objeto viene determinada por las fuerzas que actúan sobre él, y que dichas fuerzas son el resultado de las colisiones entre objetos, no podemos, en principio, encontrar una solución analítica a las ecuaciones diferenciales. Por esta razón, **la resolución de EDOs se realizará numéricamente**. Dicha resolución consistirá en tomar **pasos discretos a lo largo del tiempo** y evaluar la función f en cada paso.

Hay varios métodos para la resolución numérica de EDOs. En nuestro caso se ha implementado el **método de Euler**, ya que es el más sencillo y, por tanto, más fácil de implementar en GPU.

3.3.1. Método de Euler

Es el método numérico más sencillo. Partiendo de un estado inicial u_0 en el tiempo t_0, podemos realizar una estimación de u en un tiempo $t_0 + h$ (siendo h el tamaño del paso). Dicha estimación viene dada por:

$$u(t_0 + h) = u_0 + h\frac{du}{dt}(t_0) \tag{3.14}$$

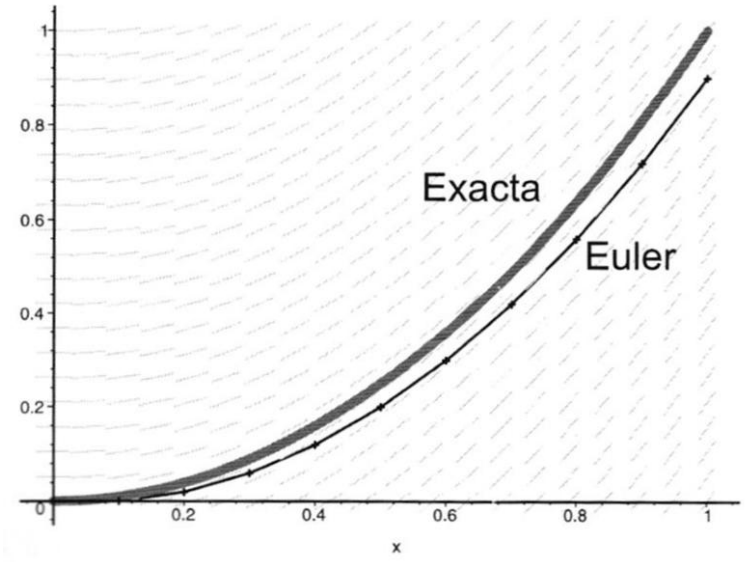


Figura 3.6: Visualización del método de Euler con $h = 0{,}1$

En la Figura 3.6 podemos observar una comparación de la solución obtenida por el método de Euler con la función obtenida analíticamente en la ecuación $du/dt = 2t$. El método de Euler con valores de h grandes no es muy preciso. Si disminuimos el tamaño del paso de simulación, podemos obtener resultados más exactos. Sin embargo, tamaños de paso demasiado pequeños pueden hacer que la simulación sea poco eficiente. Un buen método de resolución de EDOs debería permitir tamaños de paso mayores, sin perder eficiencia.

Para saber de qué modo podemos mejorar este método, tenemos que averiguar previamente el error producido. Suponiendo u continua, podemos expresar su valor, al final del paso de simulación, como una serie de Taylor:

$$u(t_0 + h) = u(t_0) + h\frac{du(t_0)}{dt} + \frac{h^2}{2!}\frac{d^2u(t_0)}{dt^2} + \cdots + \frac{h^i}{i!}\frac{d^iu(t_0)}{dt^i} + \cdots \tag{3.15}$$

Si nos quedamos con los dos primeros términos de la serie y descartamos el resto obtenemos la ecuación 3.14. El *error* cometido es igual a la diferencia entre la serie de Taylor completa y el paso de Euler. Vemos que dicho error es del orden del tamaño del paso elevado al cuadrado ($\mathcal{O}(h^2)$). Existen métodos que minimizan el error añadiendo más sumandos de la serie de Taylor, como es el caso del método de Runge-Kutta [Bar97b].

3.4. Detección de Colisiones

3.4.1. Bounding Boxes

La técnica de las *aligned bounding boxes* (cajas delimitadoras) consiste en delimitar los objetos mediante rectángulos alineados con los ejes de la escena que permiten **detectar** las posibles colisiones de forma muy sencilla. Concretamente, antes de aplicar a un par de objetos los algoritmos de colisión específicos que veremos en la sección 3.4.3, comprobamos si se produce colisión entre sus respectivas bounding boxes, transcurrido un paso de tiempo. Si no se produce dicha colisión, entonces ese par de objetos **quedan descartados** como candidatos a colisionar. De esta forma evitaremos recurrir a los algoritmos específicos, sobre aquellas parejas que sabemos que no van a colisionar, y por lo tanto, ganaremos eficiencia. Como ya veremos más adelante, han sido implementadas dos versiones del proyecto. En la primera versión, los objetos son círculos. En este caso se ha considerado inapropiado utilizar bounding boxes, ya que la comprobación de si dos círculos colisionan tiene poco coste computacional. En la segunda versión los objetos son polígonos regulares, y aquí sí se ha hecho uso de la técnica de las bounding boxes.

3.4.2. Método de Bisección

Una vez descartados los objetos que no intersecarán en el próximo paso de simulación, podemos examinar cada par de objetos candidatos a colisionar mediante algoritmos que comprueben si los objetos colisionan realmente o no. En este trabajo se ha optado por utilizar los siguientes algoritmos:

- **Comprobación de colisión entre círculos:** en este caso se utilizará el algoritmo descrito en la sección 3.4.3.

- **Comprobación de colisión entre polígonos convexos:** En este caso se utilizará una aproximación distinta al algoritmo que busca los puntos de colisión presentado en 3.4.3. Dos polígonos convexos A y B estarán solapados, y por tanto en contacto, si alguno de los vertices de A está dentro de B o si alguno de los vértices de B se encuentra dentro de A. De está forma es fácil determinar la existencia de colisión. Puede verse cómo funciona esta idea en el algoritmo en 3.1.

Sin embargo, para este proceso no sólo resulta relevante la presencia o ausencia de colisiones; también es necesario determinar el **tiempo de colisión** entre cada par de objetos. El método que proponemos en esta sección, para este fin, recibe el nombre de **bisección temporal**.

```
proc EstaDentroPuntoDePoligono(punto, poligono)
    dentro = true;
    arista = poligono.PrimeraArista();
    mientras arista! = NULL AND dentro hacer
        si (!EstaPuntoDerechaDeArista(punto, arista))
            dentro = false;
        fsi
        arista = poligono.SiguienteArista();
    fmientras
    devolver dentro;
fproc

proc EstaDentroPoligonoDePoligono(poligonoA, poligonoB)
    dentro = false;
    vertice = poligonoA.PrimerVertice();
    mientras vertice! = NULL AND !dentro hacer
        si (EstaDentroPuntoDePoligono(vertice, poligonoB))
            dentro = true;
        fsi
        vertice = poligonoA.SiguienteVertice();
    fmientras
    devolver dentro;
fproc

proc Interpenetran(poligonoA, poligonoB)
    dentro = false;
    si (EstaDentroPoligonoDePoligono(poligonoA, poligonoB)
        OR EstaDentroPoligonoDePoligono(poligonoA, poligonoB))
            dentro = true;
        fsi
    devolver dentro;
fproc
```

Algoritmo 3.1: Comprobación de existencia de colisión entre polígonos convexos

Sea t_0 el tiempo de escena actual y Δt el tiempo de paso de simulación que se quiere avanzar. Sean A y B dos objetos. Si avanzamos el estado de A y B a un tiempo $t_0 + \Delta t$ y comprobamos mediante el detector de colisiones correspondiente que *no* interpenetran, podemos ignorarlos. En caso contrario, tendremos que **restaurar** el estado de los objetos a t_0 y volver a avanzarlo, pero esta vez sólo la mitad del paso (esto es, hasta $t_0 + \Delta t/2$). En este punto:

- Si el intervalo de tiempo es menor a un cierto valor (TOL) se considerará que los objetos están colisionando y se devolverá `tmin` (para evitar la interpenetración).
- Si A y B están interpenetrando, tendremos que retroceder otra vez al tiempo t_0 y volver a avanzar el estado hasta $t_0 + \Delta t/4$ y repetir el proceso. El tiempo de colisión queda acotado en el intervalo $(t_0, t_0 + \Delta t/2)$
- Si A y B se encuentran separados, tendremos que avanzar hasta $t_0 + 3\Delta t/4$ y repetir el proceso. El tiempo de colisión queda acotado en el intervalo $(t_0 + \Delta t/2, t_0 + \Delta t)$.

De este modo podemos acotar sucesivamente el tiempo de colisión hasta que el intervalo sea lo suficientemente pequeño. En el Algoritmo 3.2 se describe el proceso completo. En la Figura 3.7 se muestra un ejemplo de cálculo de tiempo de colisión para una esfera y una pared fija.

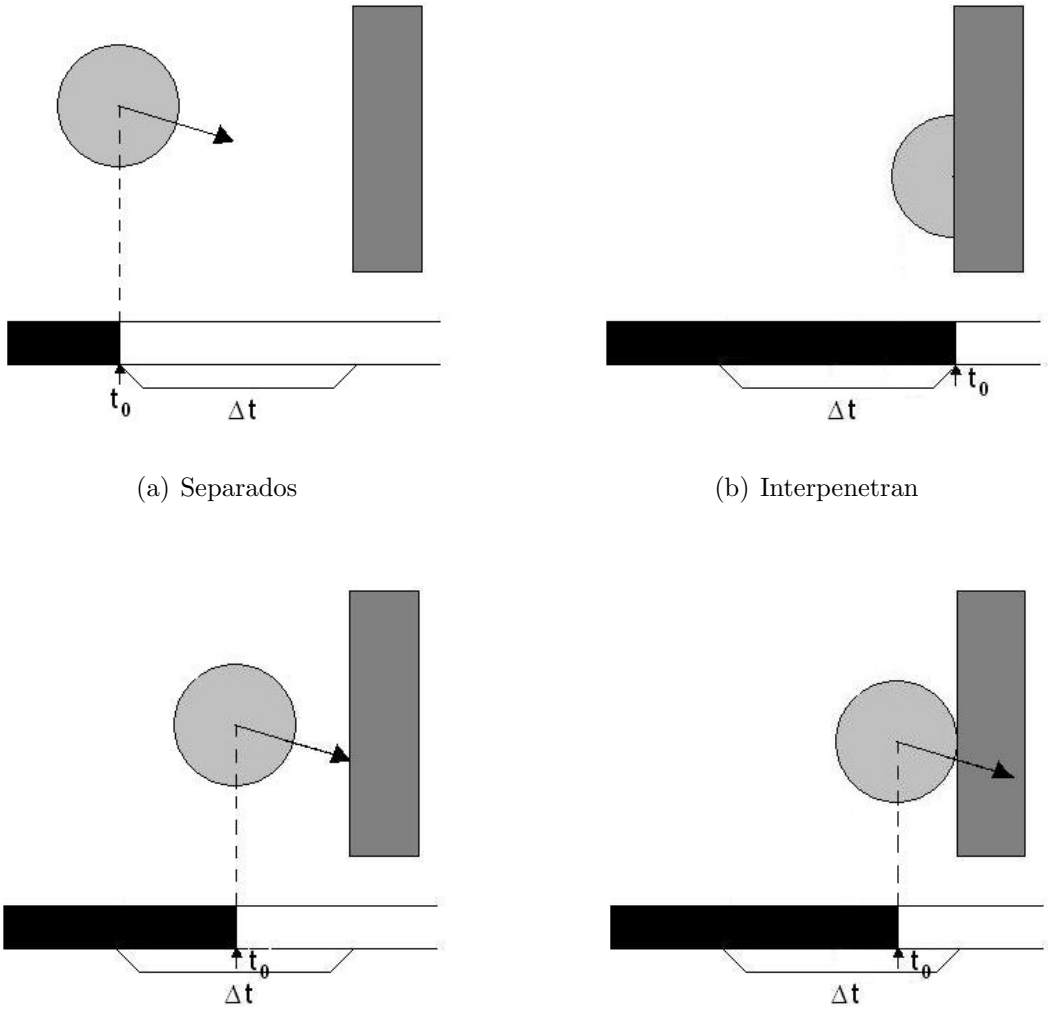
3.4.3. Algoritmos de Colisión

Una vez que hemos descartado los pares de objetos que no colisionan haciendo uso de bounding boxes, tendremos que comprobar si los objetos no descartados realmente colisionan. Como hemos visto anteriormente, para ello usaremos el algoritmo de bisección. Dicho algoritmo nos devuelve el tiempo de colisión entre un par de objetos. Una vez que tenemos todos los tiempos de colisión buscamos el tiempo mínimo y calculamos el impulso para los objetos que colisionan en ese preciso instante. Para calcular el impulso es necesario saber los puntos y normales de contacto, ya que dichos puntos serán los puntos de aplicación de la fuerza debida al impulso. En esta sección veremos cómo detectar los puntos de contacto para las geometrías usadas en el proyecto.

A partir de ahora, para todos nuestros algoritmos, debemos tener en cuenta que, para evitar errores de redondeo en las operaciones, introducimos un umbral de tolerancia $\boldsymbol{TOL}$, cuyo valor será menor que 0.01. Los valores por debajo de dicha constante se considerarán 0. Del mismo modo, los valores cuya diferencia sea menor que TOL se considerarán iguales.

```
proc CalcularTiempoColision(poligonoA, poligonoB, t0, Δt)
    tmin = t0;   tmax = t0 + Δt;
    resultado = −1;
    colisionInicio = Interpenetran(poligonoA, poligonoB);
    //Avanzar el estado de los objetos hasta tmax
    poligonoAaux = avanzar(poligonoAaux, tmax);
    poligonoBaux = avanzar(poligonoBaux, tmax);
    colisionFin = Interpenetran(poligonoAaux, poligonoBaux);
    si NOT colisionInicio AND colisionFin entonces
        mientras tmax − tmin > TOL hacer
            tmedio = (tmax − tmin)/2;
            //Avanzar el estado de los objetos hasta tmedio
            poligonoAaux = avanzar(poligonoAaux, tmedio);
            poligonoBaux = avanzar(poligonoBaux, tmedio);
            colisionMedio = Interpenetran(poligoAaux, poligoBaux);
            si NOT colisionMedio entonces
                tmax = tmedio;
            sino
                tmin = tmedio;
            fsi
        fmientras
        resultado = tmin;
    fsi
    devolver resultado;
fproc
```

Algoritmo 3.2: Método de bisección



(a) Separados

(b) Interpenetran

(c) Separados

(d) Contacto

Figura 3.7: Cálculo del tiempo de colisión para una esfera y una pared fija

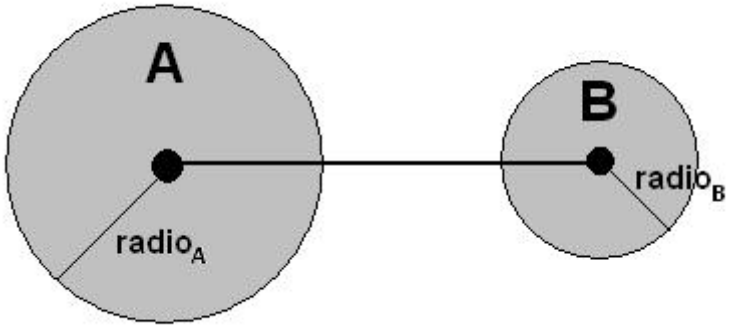


Figura 3.8: Colisión entre esferas (Como la distancia entre los centros es mayor que la suma de los radios, las esferas no están colisionando)

Colisión entre esferas

Éste es el algoritmo de colisón más sencillo. Una primera idea sería calcular la distancia que hay entre los centros de las esferas y compararla con la suma de los radios (Figura 3.8). No obstante, al calcular la distancia entre dos puntos (los centros), es necesario utilizar una raíz cuadrada, que computacionalmente es bastante costosa de realizar. Para mejorar la eficiencia del algoritmo, eliminaremos esta raíz.

Dadas las posiciones de las esferas X_a y X_b y sus respectivos radios r_a y r_b, la intersección existe si se cumple que:

$$|X_a - X_b| \leq r_a + r_b$$

Elevando al cuadrado, esto equivale a:

$$(X_{ax} - X_{bx})^2 + (X_{ay} - X_{by})^2 + (X_{az} - X_{bz})^2 \leq (r_a + r_b)^2$$

Con lo que hemos eliminado la raíz cuadrada. Si llamamos D a la parte izquierda de la ecuación, es decir, $D = (X_{ax} - X_{bx})^2 + (X_{ay} - X_{by})^2 + (X_{az} - X_{bz})^2$ y R a la parte derecha, $R = (r_a + r_b)^2$, concluimos que:

- Si $D = R$ (con tolerancia TOL), entonces se produce colisión entre ambas esferas. El punto de contacto estará en la recta que une los centros y es muy sencillo deducirlo a partir de los radios.
- Si $D < R$, entonces se produce interpenetración entre las esferas.
- Si $D > R$, entonces no se produce ni colisión ni interpenetración.

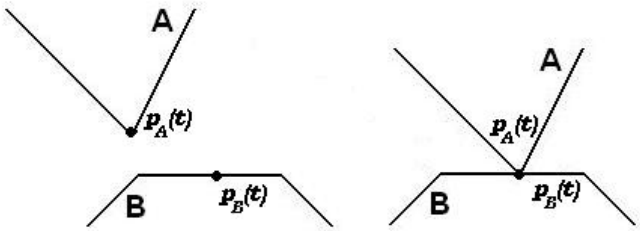


Figura 3.9: Puntos p_A y p_B de contacto

Colisión entre polígonos convexos

La idea básica de este algoritmo se basa en que todo punto de colisión entre dos polígonos corresponde a algún vértice. Por ello, teniendo dos polígonos A y B, se harán dos comprobaciones; en la primera se buscarán los vértices de A que están contenidos en alguna arista de B, y en la segunda se hará la comprobación complementaria, es decir, buscar los vértices de B que están contenidos en alguna arista de A. De esta forma se obtienen todos los puntos de contacto. Para cada punto de contacto es necesario conocer la normal de contacto, que es el vector perpendicular a la arista en la que está contenido el punto de contacto, y por convenio, debe apuntar hacia el objeto al que pertenece la arista.

3.5. Respuesta a las Colisiones. Impulso

3.5.1. Introducción

Mediante las rutinas de detección vistas anteriormente obtenemos todos los contactos que se producen simultaneamente cuando se produce el primero. Según son detectados estos puntos, es necesario tratarlos y aplicar una respuesta adecuada.

3.5.2. Definición y cálculo del impulso

Recordemos que un contacto tiene como elementos principales los dos cuerpos que colisionan (A y B), el punto de contacto p y la normal de contacto $\mathbf{n}$ (que debe salir de B y apuntar a A den la situación que se muestra en la Figura 3.9).

Cada contacto se producirá en un determinado instante de tiempo t_0. En ese instante un punto del objeto A ($p_A(t_0)$) y otro del objeto B ($p_B(t_0)$) serán iguales a p (Figura

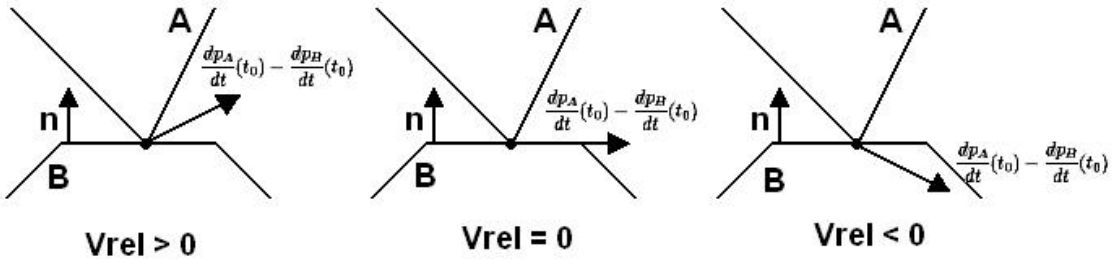


Figura 3.10: Velocidad relativa entre dos objetos

3.9). Podemos obtener la velocidad del punto en cada uno de estos objetos en el tiempo t_0 (Sección 3.2.1). Expresamos dicha velocidad como la derivada de la posición. Por tanto:

$$\frac{dp}{dt} = \mathbf{v}(t) + \omega(t) \times (p(t) - \mathbf{x}(t)) \tag{3.16}$$

Donde p es el punto del que queremos conocer la velocidad, $\mathbf{v}$ es la velocidad lineal del objeto, ω es su velocidad angular y $\mathbf{x}$ es su posición.

Aplicando la Ecuación (3.16) a los puntos $p_A(t_0)$ y $p_B(t_0)$, tenemos:

$$\frac{dp_A}{dt}(t_0) = \mathbf{v}_A(t_0) + \omega_A(t_0) \times (p_A(t_0) - \mathbf{x}_A(t_0)) \tag{3.17}$$

$$\frac{dp_B}{dt}(t_0) = \mathbf{v}_B(t_0) + \omega_B(t_0) \times (p_B(t_0) - \mathbf{x}_B(t_0)) \tag{3.18}$$

Nos interesa conocer el valor de la velocidad relativa entre los puntos $p_A(t_0)$ y $p_B(t_0)$. Ésta se define como:

$$\mathbf{v}_{rel} = \mathbf{n}(t_0) \left(\frac{dp_A}{dt}(t_0) - \frac{dp_B}{dt}(t_0) \right) \tag{3.19}$$

donde $\mathbf{n}(t_0)$ es la normal de contacto $\mathbf{n}$, ue suponemos tiene modulo o longitud 1.

El resultado de v_{rel} es un escalar. Según su valor, se pueden dar tres casos (Figura 3.10):

- $v_{rel} > 0$. En este caso los cuerpos se están alejando y por tanto, no habrá que aplicar ningún tipo de impulso.
- $v_{rel} = 0$. En este caso los cuerpos están en resting contact. En este proyecto, para este caso tampoco habrá que aplicar ningún impulso.

- $v_{rel} < 0$. Este es el caso que nos interesa. Los cuerpos colisionan (se están acercando) y por tanto será necesario modificar sus velocidades para que no interpenetren.

En este último caso, la velocidad de los cuerpos debe ser modificada instantáneamente; esto no lo podemos conseguir aplicando fuerzas. Para producir este cambio instantáneo de velocidad aplicaremos un impulso J. El impulso se puede considerar como una gran fuerza que actúa durante un corto período de tiempo:

$$\mathbf{J} = \mathbf{F}\Delta t$$

El cambio de velocidad que produce $\mathbf{J}$ puede considerarse como el cambio de velocidad que produciría $\mathbf{F}$ si la aplicásemos durante un intervalo de tiempo Δt.

La velocidad tiene una componente lineal y una angular. Vamos a ver el cambio que se produce en cada componente. El cambio en la velocidad lineal que produce $\mathbf{J}$ si se aplica a un cuerpo de masa M es:

$$\Delta\mathbf{v} = \frac{\mathbf{J}}{M} \tag{3.20}$$

Esto equivale a que el cambio del momento lineal es $\Delta\mathbf{P} = \mathbf{J}$. Al ser aplicado en un punto p, al igual que las fuerzas, produce un torque. Su valor es:

$$\Gamma_{impulso} = (p - \mathbf{x}(t)) \times \mathbf{J}$$

$\Gamma_{impulso}$ produce un cambio en el momento angular: $\Delta\mathbf{L} = \Gamma_{impulso}$. Este cambio afecta a la velocidad angular de la siguiente manera:

$$\Delta\omega = \frac{\Gamma_{impulso}}{I} \tag{3.21}$$

Ya sabemos cómo afecta un impulso cuando los cuerpos colisionan, pero todavía no sabemos cómo calcularlo. El impulso, en el caso de que no consideremos fricción, tienen la dirección de la normal de contacto $\mathbf{n}(t_0)$. Por tanto:

$$\mathbf{J} = j\mathbf{n}(t_0)$$

donde j es la magnitud del impulso. Consideraremos que el impulso actúa de forma positiva sobre el cuerpo A $(+j\mathbf{n}(t_0))$, mientras que en B actuará negativamente $(-j\mathbf{n}(t_0))$, véase la Figura 3.11.

Para las siguientes operaciones utilizaremos la siguiente notación: $\left(\frac{dp_A}{dt}(t_0)\right)^-$ es la velocidad del punto de contacto de A antes de aplicar el impulso, $\left(\frac{dp_A}{dt}(t_0)\right)^+$ es la velocidad

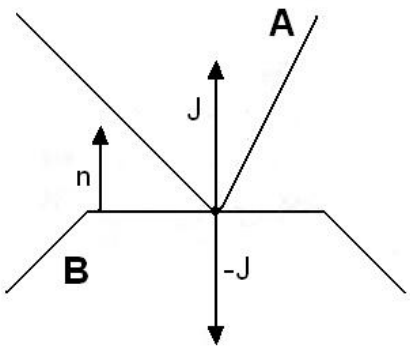


Figura 3.11: Dirección del impulso aplicado a dos cuerpos que colisionan

después de aplicarlo. Del mismo modo definimos $\left(\frac{dp_B}{dt}(t_0)\right)^-$ y $\left(\frac{dp_B}{dt}(t_0)\right)^+$. Con esta notación expresamos la velocidad relativa antes y después del impulso.

Antes del impulso:

$$\mathbf{v}_{rel}^+ = \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^+ - \left(\frac{dp_B}{dt}(t_0)\right)^+\right) \tag{3.22}$$

Después del impulso:

$$\mathbf{v}_{rel}^- = \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^- - \left(\frac{dp_B}{dt}(t_0)\right)^-\right) \tag{3.23}$$

Según leyes empíricas de colisión sin fricción se tiene que:

$$v_{rel}^+ = -\epsilon v_{rel}^- \tag{3.24}$$

Donde ϵ se conoce como **coeficiente de restitución** cuyo valor debe cumplir $0 \leq \epsilon \leq 1$. En caso de que $\epsilon = 1$ se tiene que la energía cinética tras la colisión se conserva. Si $\epsilon = 0$, la velocidad de los cuerpos tras la colisión es nula.

Para determinar el impulso que debemos aplicar tendremos que calcular el valor de j. En el resto del apartado nos dedicaremos a ello. La idea consiste en expresar v_{rel}^+ en función de v_{rel}^- y j, para poder despejarlo de la ecuación 3.24 con facilidad. Para empezar calculamos la velocidad de $p_A(t_0)$ después de la colisión en función de su velocidad antes de la colisión; lo mismo para $p_B(t_0)$. De esta forma podremos sustituir dichos valores en la ecuación 3.22.

$$\left(\frac{dp_A}{dt}(t_0)\right)^+ = \mathbf{v}_A^+ + \omega_A^+ \times \mathbf{r}_A \tag{3.25}$$

donde

$$\mathbf{r}_A = p - \mathbf{x}_A(t_0)$$

$\mathbf{v}_A^+$ y ω_A^+ son las velocidades tras aplicar el impulso. Por tanto:

$$\mathbf{v}_A^+ = \mathbf{v}_A^- + \Delta\mathbf{v} = \mathbf{v}_A^- + \frac{j\mathbf{n}(t_0)}{M_A} \tag{3.26}$$

$$\omega_A^+ = \omega_A^- + \Delta\omega = \omega_A^- + \frac{\Gamma_{impulso}}{I_A} = \omega_A^- + \frac{(\mathbf{r}_A \times j\mathbf{n}(t_0))}{I_A} \tag{3.27}$$

Sustituyendo en la ecuación (3.25) tenemos que:

$$\begin{aligned}
\left(\frac{dp_A}{dt}(t_0)\right)^+ &= \left(\mathbf{v}_A^- + \frac{j\mathbf{n}(t_0)}{M_A}\right) + \left(\omega_A^- + \frac{(\mathbf{r}_A \times j\mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A \\
&= \mathbf{v}_A^- + \omega_A^- \times \mathbf{r}_A + \frac{j\mathbf{n}(t_0)}{M_A} + \left(\frac{(\mathbf{r}_A \times j\mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A \\
&= \left(\frac{dp_A}{dt}(t_0)\right)^- + j\left(\frac{\mathbf{n}(t_0)}{M_A} + \left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A\right)
\end{aligned}$$

Realizamos los mismos cálculos para la velocidad del punto de B, pero teniendo en cuenta que el impulso tiene signo contrario ($-j\mathbf{n}(t_0)$):

$$\left(\frac{dp_B}{dt}(t_0)\right)^+ = \left(\frac{dp_B}{dt}(t_0)\right)^- - j\left(\frac{\mathbf{n}(t_0)}{M_B} + \left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right) \tag{3.28}$$

Usando estas dos últimas ecuaciones calculamos su resta, para usarla en (3.22):

$$\begin{aligned}
\left(\frac{dp_A}{dt}(t_0)\right)^+ - \left(\frac{dp_B}{dt}(t_0)\right)^+ = \left(\left(\frac{dp_A}{dt}(t_0)\right)^- - \left(\frac{dp_B}{dt}(t_0)\right)^-\right) + \\
j\left(\frac{\mathbf{n}(t_0)}{M_A} + \frac{\mathbf{n}(t_0)}{M_B} + \left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A + \left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right)
\end{aligned}$$

Ahora calculamos v_{ref}^+. Debido a que el vector $\mathbf{n}(t_0)$ tiene módulo 1, entonces $\mathbf{n}(t_0) \cdot \mathbf{n}(t_0) = 1$.

$$
\begin{aligned}
v_{rel}^{+} &= \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^{+} - \left(\frac{dp_B}{dt}(t_0)\right)^{+}\right) \\
&= \mathbf{n}(t_0)\left(\left(\left(\frac{dp_A}{dt}(t_0)\right)^{-} - \left(\frac{dp_B}{dt}(t_0)\right)^{-}\right)\right. \\
&+ \left. j\left(\frac{\mathbf{n}(t_0)}{M_A} + \frac{\mathbf{n}(t_0)}{M_B} + \left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A \left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right)\right) \\
&= v_{rel}^{-} + j\left(\frac{1}{M_A} + \frac{1}{M_B} + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right)
\end{aligned}
$$

Hemos conseguido expresar v_{rel}^{+} en función de j y v_{rel}^{-}. Sustituiremos v_{rel}^{+} en (3.24):

$$
\mathbf{v}_{rel}^{-} + j\left(\frac{1}{M_A} + \frac{1}{M_B} + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right) = -\epsilon \mathbf{v}_{rel}^{-} \tag{3.29}
$$

Por último, podemos despejar j:

$$
j = \frac{-(1+\epsilon)\mathbf{v}_{rel}^{-}}{\left(\frac{1}{M_A} + \frac{1}{M_B} + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_A \times \mathbf{n}(t_0))}{I_A}\right) \times \mathbf{r}_A + \mathbf{n}(t_0)\left(\frac{(\mathbf{r}_B \times \mathbf{n}(t_0))}{I_B}\right) \times \mathbf{r}_B\right)} \tag{3.30}
$$

Capítulo 4

Implementación

4.1. Introducción

A lo largo de este capítulo se explicará cómo se han implementado las dos versiones del proyecto. Hemos definido una serie de clases, comunes a ambas, que proporcionan métodos para crear y usar shaders, y que facilitan el uso de texturas como flujo de datos. Estas clases son las siguientes:

- GLSLShader
- GLSLProgram
- Stream
- Kernel

En este capítulo también se analizarán aspectos de la implententación específicos de cada versión, como son el bucle de simulación y el flujo de datos que tiene lugar entre shaders, y entre cada shader y la CPU.

4.2. GLSLShader

Esta clase tiene como objetivo encapsular la funcionalidad que ofrece GLSL en lo que se refiere a la creación y gestión de objetos shader. De esta forma resultará más fácil y menos engorroso usar las funciones necesarias para dicho fin. Los principales métodos de la clase son:

- `GLSLShader(const std::string &filename,`
  `GLenum shaderType = GL_VERTEX_SHADER);`

  Esta función es un constructor que permite especificar el archivo donde se encuentra el código del shader, e indicar su tipo. En caso de que no indiquemos el tipo, se considerará, por defecto, que es un shader de vértices.

- `GLSLShader(GLenum shaderType = GL_VERTEX_SHADER);`

  Esta función es otro constructor. En este caso crea un shader vacío, y especifica su tipo. Al igual que el otro constructor, el tipo por defecto es shader de vértices.

- `~GLSLShader();`

  La función anterior es un destructor. Será llamado cuando el objeto sea borrado.

- `void compile();`

  Al llamar a compile(), se está pidiendo que el objecto shader sea compilado, es decir que se compile el código que tiene asociado. Dicho código ha sido asignado a través del constructor o a través de la función `setShaderSource(std::string &code)`.

- `bool isCompiled() const;`

  Llamando a esta función podremos comprobar si la compilación se ha realizado correctamente o no.

- `void getShaderLog(std:string &log) const;`

  En caso de que se hayan producido errores a la hora de compilar, estos errores son guardados en el registro del shader. Podemos acceder a ellos a través de esta función.

- `void getShaderSource(std:string &shader) const;`

  Devuelve el código que tiene asociado el shader actualmente.

- `void setShaderSource(std:string &code);`

  Reemplaza el código que tiene asociado el shader actualmente.

- `GLuint getHandel() const;`

  Nos permite obtener el manejador de objeto shader. En GLSL todos los objetos se gestionan mediante un manejador, que no es otra cosa que un número entero que lo identifica.

- `void getParameter(GLenum param, GLint *data) const;`

  Permite consultar cierta información del shader en función del parámetro de entrada `param`. A continuación indicamos el resultado correspondiente a los principales valores de `param`:

- GL_SHADER_TYPE: data devuelve GL_VERTEX_SHADER si es el shader es un shader de vértices, y GL_FRAGMENT_SHADER si es un shader de fragmentos.
- GL_DELETE_STATUS: data devuelve GL_TRUE si el shader está actualmente marcado para ser eliminado, y GL_FALSE en otro caso.
- GL_COMPILE_STATUS: data devuelve GL_TRUE si la última operación de compilación en el shader se realizó satisfactoriamente, y GL_FALSE en cualquier otro caso.
- GL_INFO_LOG_LENGTH: data devuelve el número de caracteres que hay en el registro de información del shader, incluyendo el carácter de terminación (es decir, el tamaño del buffer de caracteres necesario para almacenar el registro de información). Si el shader no tiene registro de información, se devuelve 0.
- GL_SHADER_SOURCE_LENGTH: data devuelve la longitud de la concatenación de los strings que componen el código del shader, incluyendo el carácter de terminación (es decir, el tamaño del buffer de caracteres necesario para almacenar todo el código del shader). Si no existe código, se devolverá 0.

4.3. GLSLProgram

De forma similar a la clase GLSLShader, el objetivo de esta clase es ocultar la funcionalidad que ofrece GLSL en lo que se refiere a la creación y gestión de objetos programa. Las funciones públicas de GLSLProgram son las siguientes:

- `GLSLProgram();`

  Esta función es un constructor que crea un programa vacío. Por tanto la única acción que lleva a cabo es generar un manejador para el objeto programa.

- ```
  GLSLProgram(const std::string &shader,
              unsigned int shaderType=GL_VERTEX_SHADER_ARB);
  ```

  Este segundo constructor carga el archivo especificado por la variable shader, lo compila y lo enlaza para que el shader esté listo para ser usado. La variable `shaderType` indica el tipo de shader que vamos a añadir al programa.

- ```
  GLSLProgram(const std::string &vertexShader,
              const std::string &fragmentShader);
  ```

  Este último constructor es similar al visto anteriormente, pero la diferencia es que se enlazan, al programa, dos tipos de shader.

- `~GLSLProgram();`

  La función anterior es un destructor para borrar el objeto programa. Además se encargará de eliminar, previamente, todos los objetos shader que el objeto programa tenga asociados.

- `void attach(GLSLShader &shader);`
- `void attach(GLSLShader *shader);`

  Estas funciones permiten asociar un objeto shader al objeto programa.

- `void detach(GLSLShader &shader);`
- `void detach(GLSLShader *shader);`

  Estas funciones permiten romper la asociación de un objeto shader al objeto programa.

- `void link();`

  Enlaza, al objeto programa, todos los objetos shader que tenga asociados.

- `void use() const;`

  Activa el objeto programa para que sea usado. De este modo los shader enlazados al programa sustituirán la funcionalidad predefinida que tiene lugar en las etapas de procesamiento de vértices y de fragmentos.

- `void disable() const;`

  Desactiva el objeto programa para que deje de ser usado. De esta forma se vuelve a utilizar funcionalidad predefinida de la tubería gráfica.

- `void sendUniform(const std::string &name, float x);`
- `void sendUniform(const std::string &name, float x, float y);`
- `void sendUniform(const std::string &name, float x, float y, float z);`
- ```
  void sendUniform(const std::string &name, float x, float y, float z,
                   float w);
  ```
- `void sendUniform(const std::string &name, int x);`
- `void sendUniform(const std::string &name, int x, int y);`
- `void sendUniform(const std::string &name, int x, int y, int z);`
- `void sendUniform(const std::string &name, int x, int y, int z, int w);`
  ```

- `void sendUniform(const std::string &name, float *m, bool transp=fasle, int size=4);`

  Todas las funciones anteriores sirven para actualizar el valor de variables uniformes. Toman el nombre de la uniforme, determinan cuál es el slot o localizador donde se almacena la uniforme, y escriben allí los nuevos datos. La búsqueda de un localizador por parte de un objeto programa es un proceso bastante costoso, por ello estas funciones usan un sistema de caché, para que sólo haya coste la primera vez o después de volver a enlazar. Si el nombre de la uniforme no puede ser encontrado, entonces se lanza una excepción std::logic_error() con los detalles de qué uniforme no pudo encontrarse.

- `void sendUniform(GLuint location, float x);`

- `void sendUniform(GLuint location, float x, float y);`

- `void sendUniform(GLuint location, float x, float y, float z);`

- `void sendUniform(GLuint location, float x, float y, float z, float w);`

- `void sendUniform(GLuint location, int x);`

- `void sendUniform(GLuint location, int x, int y);`

- `void sendUniform(GLuint location, int x, int y, int z);`

- `void sendUniform(GLuint location, int x, int y, int z, int w);`

- `void sendUniform(GLuint location, float *m, bool transp=fasle, int size=4);`

  Este nuevo grupo de instrucciones tienen la misma función que el grupo de instrucciones `sendUniform` visto antes. La única diferencia es que se les pasa directamente la localización de la variable en lugar del nombre. Cuando se usan estas funciones hay que estar seguro de que el número del slot es correcto.

- `GLuint getUniformLocation(const std::string &name) const;`

  Devuelve la localización de una variable uniforme del programa; en caso de que no exista el nombre de esa varible devuelve -1.

- `GLuint setAttributeLocation(const std::string &name, GLuint location);`

  Permite seleccionar la localización de una variable atributte asociada al objeto programa. La nueva localización seleccionada no tendrá efecto hasta que no se lleve a cabo una operación de enlazado (link).

- `GLuint getAttributeLocation(const std::string &name) const;`

  Devuelve la localización de una variable attribute asociada al programa.

- `void getShaderLog(std:string &log) const;`

  En caso de que se hayan producido errores al realizar alguna operación sobre el objeto programa, estos errores se guardan en el registro del programa. Podemos acceder a ellos a través de esta función.

- `GLuint getHandel() const;`

  Nos permite obtener el manejador del objeto programa.

- `void getParameter(GLenum param, GLint *data) const;`

  Permite consultar cierta información del objeto programa en función del parámetro de entrada `param`. A continuación mostramos el resultado correspondiente a los principales valores de `param`:

  - GL_DELETE_STATUS: data devuelve GL_TRUE si el programa está actualmente marcado para ser eliminado, y GL_FALSE en otro caso.
  - GL_LINK_STATUS: data devuelve GL_TRUE si la última operación de enlace sobre el programa tuvo éxito, y GL_FALSE en otro caso.
  - GL_VALIDATE_STATUS: data devuelve GL_TRUE si la última operación de validación sobre el programa tuvo éxito, y GL_FALSE en otro caso.
  - GL_INFO_LOG_LENGTH: data devuelve el número de caracteres que hay en el registro de información del objeto programa, incluyendo el carácter de terminación (es decir, el tamaño de el buffer de caracteres necesario para almacenar todo el registro de información). Si el programa no tiene registro de información, se devuelve 0.
  - GL_ATTACHED_SHADERS: data devuelve el número de objetos shader asociados al objeto programa.
  - GL_ACTIVE_ATTRIBUTES: data devuelve el número de variables atributo activas en el objeto programa.
  - GL_ACTIVE_ATTRIBUTES_MAX_LENGHT: data devuelve la longitud del nombre más largo de los atributos activos en el objeto programa, incluyendo el carácter de terminación (es decir, el tamaño del buffer de caracteres necesario para almacenar el nombre del atributo más largo). Si no hay atributos activos, devuelve 0.
  - GL_ACTIVE_UNIFORMS: data devuelve el número de variables uniformes activas en el objeto programa.

  - GL_ACTIVE_UNIFORM_MAX_LENGHT: data devuelve la longitud del nombre más largo de las uniformes activas en el objeto programa, incluyendo el carácter de terminación (es decir, el tamaño del buffer de caracteres necesario para almacenar el nombre de la uniforme más largo). Si no hay atributos activos, devuelve 0.

- `void validate() const;`

  Se llama a esta función para pedir a OpenGL que valide el objeto programa.

- `bool isValidProgram() const;`

  Devuelve un boolean para indicar si el objeto programa es válido o no.

- `void getAttributeInfo(GLuint location, std::string &name, GLenum &type, GLint &size) const;`

- `void getUniformInfo(GLuint location, std::string &name, GLenum &type, GLint &size) const;`

  Las dos funciones anteriores obtienen información sobre las variables uniformes o atributos, devolviéndola en las variables proporcionadas.

- `void getAttachedShader(std::vector<GLuint> &shaderhandles);`

- `void getAttachedShader(std::vector<GLSLShader> &shaders);`

  Estas dos últimas funciones permiten conocer qué shaders tiene asociados el objeto programa. Ambas funciones requieren un std::vector como parámetro de salida para devolver la información. La primera versión sólo devuelve los manejadores de los objetos shader. La segunda, el vector de GLSLShader correspondiente a los manejadores.

4.4. Stream

Ya se ha visto que la forma de establecer la comunicación shader-shader o shader-CPU es a través de texturas. Para facilitar todas las operaciones que se deben realizar sobre las texturas, como son la creación, la carga y la lectura de datos, se ha implementado la clase `Stream`. Cada objeto de esta clase tendrá asociada una o dos texturas, que serán el objetivo de la mayoría de las funciones implementadas.

La clase `Stream` es abstracta y de ella heredan `StreamSimple` y `StreamDoble`, como se puede ver en la Figura 4.1. El motivo de esta distinción se basa en el uso de la técnica ping-pong vista en la sección 2.5.4. Como ya vimos, dicha técnica consiste en el uso de dos texturas en el caso de que los datos sean de entrada y de salida. Por tanto, la clase

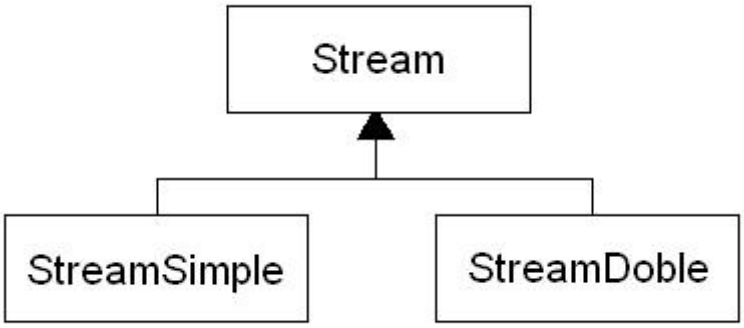


Figura 4.1: Jerarquía de la clase Stream

`StreamSimple` será usada para datos que sean únicamente de entrada o de salida, y tendrá asociada sólo una textura; mientras que la clase `StreamDoble` se usará en el caso de que los datos sean de entrada y de salida, y tendrá asociada dos texturas.

A continuación se muestran las funciones y variables pertenecientes a la clase `Stream`:

- Variables:
  - identificadores de textura: En el caso de `StreamSimple` se guarda un solo identificador, ya que sólo tiene asociada una textura. Para `StreamDoble` se tienen dos identificadores, uno para la textura de lectura y otro para la de escritura.
  - `int ancho`: se corresponde con el ancho de la textura asociada.
  - `int alto`: se corresponde con el alto de la textura asociada.
  - `GLint FORMATO_INTERNO`: Esta variable indica el formato interno usado en la textura y depende de la tarjeta utilizada. En el proyecto se ha utilizado una tarjeta NVIDIA, por lo que los formatos internos posibles son los siguientes: GL_FLOAT_R32_NV, GL_FLOAT_RGB32_NV y GL_FLOAT_RGBA32_NV.
  - `GLint FORMATO_PIXEL`: Esta variable indica el formato de textura. Como ya vimos en la sección 2.5.2, hay varios formatos posible. La clase `Stream` permite usar los siguientes formatos: GL_LUMINANCE, GL_RGB y GL_RGBA.
- Funciones:
  - `Stream(int nancho,int nalto,TipoFormato formato);`
    TipoFormato es la siguiente enumeración:

    ```
    enum TipoFormato {LUMINANCE,RGB,RGBA};
    ```

Para `StreamSimple`, el constructor crea una textura de tamaño nancho×nalto. El formato de la textura y el formato interno son seleccionados en función del parámetro `formato`. Esta textura es asociada al `StreamSimple` a través de la variable `id` que contiene el identificador de la textura correspondiente. En cambio, para `StreamDoble`, el constructor funciona de forma similar, pero crea dos texturas, una para lectura y otra para escritura.

- `int getIdEscritura();`
  `int getIdLectura();`
  Devuelven el identificador de la textura de escritura o de lectura. En el caso de `StreamSimple`, devuelve el mismo identificador.
- `void intercambiar();`
  Se intercambia la textura de escritura por la de lectura y viceversa. En el caso de `StreamSimple` no hace nada.
- `void cargarDatos(GLvoid* datos);`
  Carga en la textura de lectura los datos contenidos en el parámetro de entrada.
- `void extraerDatos(GLvoid* datos);`
  Extrae los datos contenidos en la textura de lectura y los guarda en el parámetro de salida que pasamos a la función.

4.5. Kernel

Podemos definir un `Kernel` como una etapa del flujo de datos, global del programa, y por ello tendrá unos ciertos datos de entrada y de salida. Estos datos son almacenados en objetos de la clase `Stream`. Como consecuencia, un `Kernel` tiene una serie de streams de entrada y de streams de salida. Esto puede verse en la Figura 4.2.

Para el procesamiento o flujo de datos, la clase `Kernel` hace uso de la GPU. Por este motivo cada objeto `Kernel` dispone de una referencia a un `GLSLProgram` y a dos `GSLSShader` (una para el shader de vértices y otra para el shader de fragmentos). Haciendo uso de estas referencias, la clase `Kernel` consigue reemplazar el funcionamiento predefinido de la tarjeta por otro distinto que nos permita realizar las operaciones deseadas.

A continuación se muestran las variables y funciones pertenecientes a la clase `Kernel`:

- Variables:
  - `vector <Stream*> streamsIn`: Streams de entrada.
  - `vector <Stream*> streamsOut`: Streams de salida.
    Si un stream es de entrada y salida, debe ser incluido en los dos vectores.

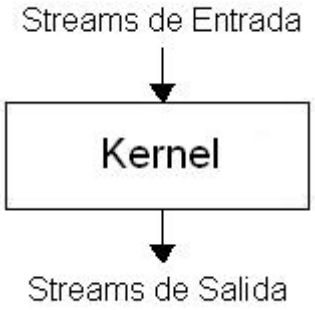


Figura 4.2: Entrada y Salidad de un kernel

- `GLSLProgram* programa`: objeto programa asociado al kernel.
- `GLSLShader* vertexShader`: shader de vértices asociado al kernel.
- `GLSLShader* fragmentShader`: shader de fragmentos asociado al kernel.
- `GLuint fbo`: Identificador del FrameBuffer utilizado para dirigir el flujo de salida hacia la textura deseada.
- `int ancho`: ancho de los Streams de salida.
- `int alto`: alto de los Streams de salida. Lo normal es que todos los streams de salida tengan el mismo tamaño. De no ser así, se debería usar el ancho y alto más grande de entre los streams de salida.

- Funciones:

  - ```
    Kernel(int nancho, int nalto, char* nombreLog, Tipo tipoShader,
           string shader);
    ```

    Tipo es la siguiente enumeración:

    ```
    enum  Tipo {VERTEX_SHADER,FRAGMENT_SHADER};
    ```

    Este constructor crea un kernel con un solo shader, el tipo de shader está determinado por el parámetro `tipoShader`. El código del shader se encuentra en el archivo especificado por el parámetro shader. El parámetro `nombreLog` indica el fichero en el que se guardarán las incidencias que se produzcan al compilar y enlazar el shader. El tamaño del kernel viene determinado por `nancho` y `nalto`. El constructor realiza los pasos necesarios para usar los shader, estos pasos los vimos en la sección 2.4.3: crear el objeto shader, cargar el código en él, compilarlo, crear el objeto programa y enlazar el shader al programa.
    ```

- `Kernel(int nancho, int nalto, char* nombreLog, string vShader, string fShader);`
  Este otro constructor es similar al anterior, la diferencia es que crea un kernel con dos shader. El código del shader de vértices se encuentra en el archivo `vShader` y el del shader de fragmentos en `fShader`.
- `void iniciarKernel();`
  Se encarga de inicializar ciertos aspectos de los shader, como por ejemplo el valor de las uniformes. Esta función se llama dentro de los constructores.
- `void addInput(Stream* entrada);`
  Añade el stream entrada al vector `streamsIn`.
- `void addOutput(Stream* salida);`
  Añade el stream salida al vector `streamsOut`.
- `void desactivarKernel();`
- `void activarKernel();`
  Activa o desactiva el objeto programa. Si el programa esta activo los shaders serán usados, sino se usará la funcionalidad predefinida de la tarjeta.
- `void iniciarPasada();`
  Configura el puerto de vista y prepara las texturas de entrada y de salida. También es posible que dé valores a las variables uniform.
- `void pasada();`
  Se realiza el flujo de datos y, para los streams de entrada/salida, se aplica el intercambio de texturas de lectura y escritura.

La clase `Kernel` es abstracta y por tanto las clases que hereden de ella deben implementar algunas de las funciones anteriores. Esto es así porque no todos los kernels tienen las mismas variables uniformes, ni realizan el flujo de datos de la misma forma.

En las siguientes secciones veremos cómo han sido implentadas las dos versiones existentes, veremos las etapas del bucle de simulación, y cómo cada una de estas etapas se corresponde con un Kernel.

4.6. Versión 1

En esta versión los objetos utilizados para la simulación son **círculos**. Usando este tipo de geometría se consigue simplificar significativamente el flujo de datos, y por tanto la adaptación de dicho flujo para, su ejecución en GPU no resultará demasiado complicada.

Podemos distinguir las siguientes etapas en el bucle de simulación:

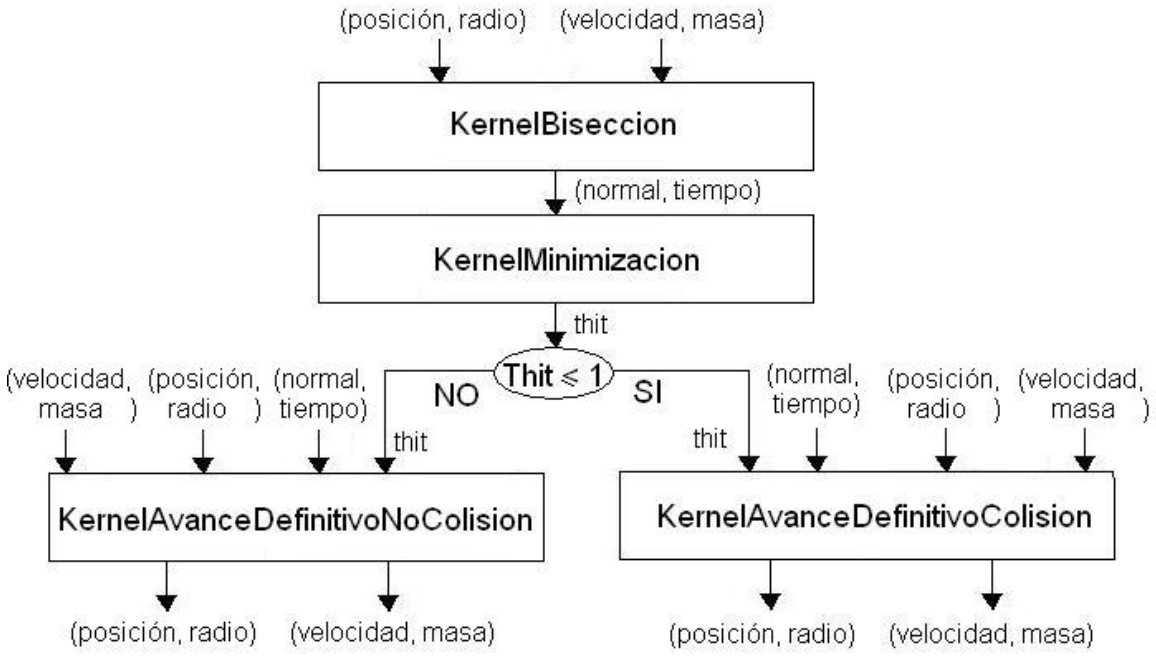


Figura 4.3: Diagrama de flujo de datos entre los kernels de la primera versión

- Buscar los pares de objetos que colisionan en un intervalo de tiempo. Para cada par de objetos calcular la normal de contacto y el tiempo de impacto exacto. El punto de contacto no es necesario guardarlo, porque podremos calcularlo fácilmente a partir de la normal. Para buscar el tiempo se utiliza el método de bisección visto en la sección 3.4.2.

- Buscar el tiempo en que se produce el primer impacto. Para ello aplicaremos la técnica de minimización vista en la sección 2.5.5.

- Avanzar el estado de los objetos hasta el tiempo del primer impacto. Esto supone calcular y aplicar un impulso a los objetos que colisionan.

Cada una de estas etapas será llevada por un kernel distinto. Los kernels que se han usado son los siguientes: KernelBiseccion, KernelMinimizacion, KernelDefinitivoColision y KernelAvanceDefinitivoNoColision. Los dos últimos son similares, pero el último de ellos no requiere calcular ningún tipo de impulso. Podrían haberse implementado como un único kernel, pero por motivos de eficiencia se ha decidido implementarlo de este modo.

En la Figura 4.3 se muestra el flujo de datos que se produce entre los distintos kernels.

A continuación se va a explicar el flujo de datos que tiene lugar en el proyecto y que podemos ver en la Figura 4.3:

- **Etapa de Bisección.**

  En primer lugar las posiciones, velocidades, masas y radios de los objetos son pasados al kernel de bisección a través de dos texturas. En una se pasan las posiciones junto con los radios y en la otra las velocidades con las masas. Este kernel se encargará de buscar los pares de objetos que colisionan y para cada par de objetos devolverá la normal de contacto y el tiempo de impacto. Estos datos son devueltos en el stream de normales y tiempos de tamaño n $\times$ n. Como se puede ver, cada fragmento con el que trabaja este kernel se corresponde con un par de objetos. El kernel realiza los cálculos aplicando el método de bisección visto en la sección 3.4.2, y que comprueba la existencia de interpenetración entre objetos mediante el algoritmo de colisión entre círculos de la sección 3.4.3.

- **Etapa de Minimización.**

  Una vez que se tienen todos los tiempos de impacto se busca el menor de ellos, es decir, el tiempo en que se produce el primer impacto. Para buscar el mínimo se pasa por el kernel de minimización que aplica el método que vimos en la sección 2.5.5.

- **Etapa de Avance.**

  Si el tiempo calculado en la etapa de minimización es menor al tiempo de un paso, quiere decir que al menos un par de objetos colisionarán, y por lo tanto será necesario calcular impulsos. En este caso se pasa por KernelAvanceDefinitivoColision. Este kernel tendrá como entradas las texturas correspondientes a la posición, velocidad, radio y masa. Además también se le pasará el tiempo calculado en la etapa de minimización a través de una variable uniforme. Para cada objeto comprobará si su tiempo de colisión con el resto de objetos es menor que el tiempo del primer impacto; si sucede esto calcula el impulso debido a cada impacto y lo acumula en una variable para obtener así el impulso total que actúa sobre el objeto. Una vez calculado el impulso total, lo aplica al objeto que corresponde al fragmento, modificando la textura que almacena su posición y su velocidad.

  Para el cálculo de los impulsos se usa la fórmula vista en la sección 3.5.

  En el caso de que el tiempo del primer impacto sea mayor al tiempo de un paso, no es necesario calcular impulsos, ya que no se habrá producido ninguna colisión. En este caso actuará KernelAvanceDefinitivoNoColision. Como entradas tendrá las texturas correspondientes a la posición, velocidad, radio y masa. La función de este kernel es modificar las posiciones aplicándoles la velocidad correspondiente.

Como se puede ver en la Figura 4.3 se usan tres streams para almacenar los datos:

- Stream de posición y radio. Este stream es de la subclase StreamDoble, ya que se comporta como de entrada y salida en los kernels de avance definitivo. Se utiliza una textura de tipo rgba: en las tres primeras componentes se guardan las coordenadas de la posición (aunque nos sobra la coordenada z), y en la componente alfa se almacena el radio. La textura es de tamaño n×1, donde n es el número de objetos.

- Stream de velocidad y masa. Este stream es de la subclase StreamDoble, ya que se comporta como de entrada y salida en los kernels de avance definitivo. Se utiliza una textura de tipo rgba: en las tres primeras componentes se guarda la velocidad (de nuevo sobra la coordenada z) y en la componente alfa se almacena la masa. La textura es de tamaño n×1, donde n es el número de objetos.

- Stream de normales y tiempos. Es un stream de la subclase StreamSimple, ya que en ningún kernel se comporta como de entrada y salida a la vez. La textura es de tamaño n×n, donde n es el número objetos. Se utiliza una textura de tipo rgba: en las tres primeras componetes se guarda la normal (sobra la z), y en la componente alfa se almacena el tiempo de impacto. En el texel de coordenadas (i,j) se almacena la normal de contacto y el tiempo de impacto entre el objeto i y el objeto j (En caso de no existir colisión se almacena (0,0,0,2)). Usando esta idea, el texel (i,j) y el (j,i) contendrían los mismos datos, por ello basta con procesar uno de los dos fragmentos. Aprovechando esta simetría, dentro del shader de bisección sólo realizaremos los cálculos para la diagonal superior de la textura.

4.7. Versión 2

En esta segunda versión los objetos utilizados para la simulación son **polígonos convexos**. Al usar este tipo de geometría se complica considerablemente el flujo de datos con respecto a la versión anterior, ya que se introduce rotación, se introducen bounding boxes, los test de colisión son más complejos, etc. Para ver este aumento de dificultad en la implementación podemos comparar el número de etapas que hay en el bucle de simulación para la primera versión con el número de etapas de esta nueva versión. A continuación se enumeran las etapas para esta última:

- Calcular las bounding boxes. En esta etapa se calcula, para todos los objetos, la mínima aligned bounding box que lo contiene al final del intervalo de tiempo.

- Comprobar qué aligned bounding boxes se solapan. Se comprueban las bounding boxes dos a dos para comprobar si se solapan. En caso de que se solapen, los objetos contenidos son candidatos a colisionar.

- Buscar los tiempos exactos en que se producen las colisiones. Para los pares de objetos candidatos a colisionar, se comprueba si realmente colisionan y se busca el tiempo en que se producirá la colisión. Para buscar el tiempo se utiliza el método de bisección visto en la sección 3.4.2.
- Buscar el tiempo en que se produce el primer impacto. Este Kernel será igual que el de la primera versión.
- Calcular el impulso que se produce entre dos objetos, para aquellos pares de objetos que colisionan en el mismo tiempo en que se produce el primer impacto.
- Avanzar el estado de los objetos hasta el tiempo del primer impacto. Esto supone aplicar los impulsos calculados en la etapa anterior a los objetos implicados en la colisión.

Al igual que en la primera versión, cada una de estas etapas será ejecutada por un kernel distinto. Los kernels que se han usado son los siguientes: KernelCalcularAABB, KernelAABB, KernelBiseccionEtapa1, kernelBiseccionEtapa2, KernelMinimizacion, KernelInterseccion, KernelAvanceDefinitivoColision y KernelAvanceDefinitivoNoColision.

Las tarjetas gráficas NVIDIA tienen un límite en el número de instrucciones que se pueden ejecutar en el shader de fragmentos. Por otra parte, la etapa de bisección tiene una gran carga computacional, ya que incluye un bucle para la búsqueda del tiempo. En dicho bucle se realizan operaciones de comprabación de colisión, cuyo coste está relacionado con el número de vértices de los objetos implicados. Por ello, si en el programa se tienen objetos con gran cantidad de vértices es posible que se supere el número de instrucciones que puede ejecutar un shader de fragmentos. Por este motivo hemos decidido realizar la etapa de bisección llamando de forma iterativa a la ejecución de un kernel, hasta que el tiempo sea encontrado. KernelBiseccionEtapa1 se corresponde con la primera pasada por el bucle, en el resto de pasadas se ejecutará KernelBisceccionEtapa2. Ambos kernels tienen la misma funcionalidad, pero varían en los streams que reciben.

La existencia de dos Kernels para la etapa de avance de estado (KernelAvanceDefinitivoColision y KernelAvanceDefinitivoNoColision) se debe a la misma razón que empleamos en la versión 1.

En la figura 4.4 se muestra el flujo de datos que se produce entre los distintos kernels.

A continuación vamos a explicar el flujo de datos que tienen lugar en el proyecto y que podemos ver en la Figura 4.4:

- **Calcular las aligned bounding boxes.**

  Para esta etapa usaremos únicamente el kernel de cálculo de aligned bounding boxes. Como entrada recibirá las posiciones, rotaciones, velocidades lineales, velocidades

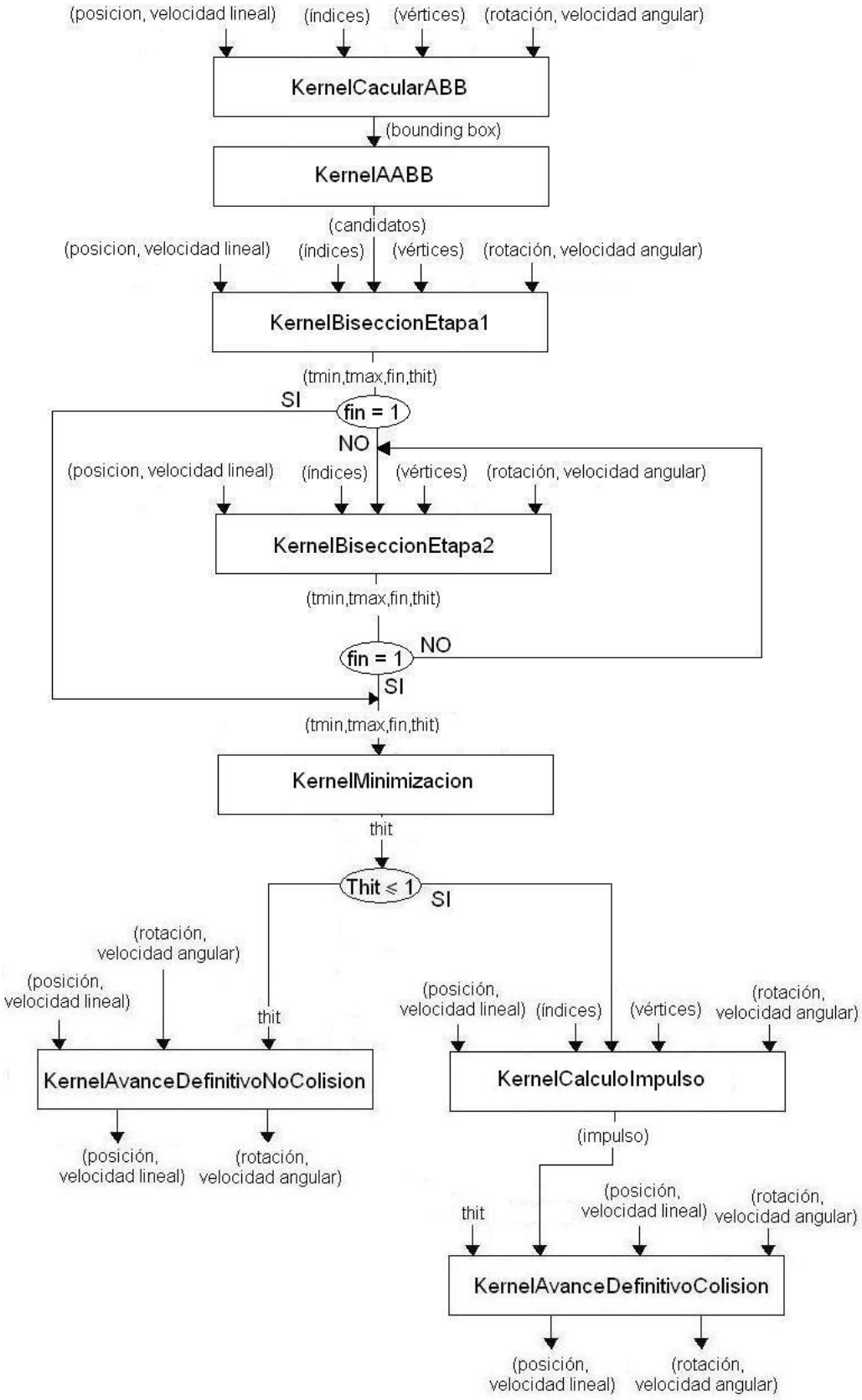


Figura 4.4: Diagrama de flujo de datos entre los kernels de la segunda versión

angulares y geometrías de los objetos. Dichos valores serán proporcionados a través de dos texturas de tamaño n × 1, donde n es el número de objetos. Una de las texturas llevara la información de la posición y la velocidad lineal, y la otra la información de la rotación y la velocidad angular. Además también se le pasará al kernel la geometría de los objetos, para ello usaremos dos texturas más, la textura de índices (de tamaño n × 1) y la textura de vértices (su tamaño depende del número de vértices de todos los objetos de la escena). Con estos datos este kernel calculará y devolverá las ABB, al final de un paso de tiempo, de todos los objetos en el stream aligned bounding boxes. Cada fragmento tratado se corresponde con un objeto, que es para quien calcularemos su bounding box. La forma de calcular la ABB consistirá en calcular la posición y rotación final del objeto y aplicárselas a sus vértices para calcular sus coordenadas globales (las coordenadas locales de los vértices son pasadas a través del stream de índices y de vértices). Una vez tenemos las coordenadas globales de todos los vértices se calcula el rectángulo que envuelve a todos los vértices. Este rectángulo se corresponde con la ABB. El resultado es devuelto en una textura de tamaño n × 1 que servirá como entrada a la siguiente etapa del bucle de simulación.

- **Comprobar el solapamiento de las aligned bounding boxes.**

  Esta etapa será ejecutada por un solo kernel, el kernel de cálculo de ABB. Una vez calculadas las aligned bounding boxes se pasan a kernelAABB a través de una textura, de tamaño n × 1, para comprobar qué pares de bounding boxes se solapan. Cada fragmento procesado se corresponde a dos objetos, y devolverá en cada fragmento si las ABB de esos dos objetos se solapan. Por lo tanto se necesitará como salida una textura bidimensional, de tamaño n × n. El texel de coordenadas (i,j) indica si los objetos i y j son candidatos a colisionar o no. Como ya se ha comentado es posible usar simetría para evitar tener que procesar todos los fragmentos.

- **Etapa de Bisección.**

  Esta etapa ha sido descompuesta en dos subetapas. En ambas etapas se pasan como entrada las texturas correspondientes a la posición, rotación, velocidad lineal, velocidad angular y geometría de los objetos. El kernel que se encarga de la primera iteración del método de bisección, kernelBiseccion1, tiene también como entrada la textura de candidatos obtenida en la etapa anterior. Con toda esta información se realiza el primer acotamiento del intervalo de tiempo. Como salida se devolverá una textura que llevará información necesaria para la siguiente iteración del método de bisección. Cada fragmento vuelve a corresponder a un par de objetos, y la información contenida en un texel es el tiempo máximo y mínimo del intervalo, si se ha terminado la búsqueda del tiempo y el tiempo de impacto encontrado entre los dos objetos.

  Una vez realizada la primera pasada por el método de bisección hay dos posibilidades: que hayamos terminado o que aún tengamos que seguir acotando el intervalo de

colisión para algún par de objetos. En el primer caso nos saltaremos este segundo kernel, en el otro caso tendremos que pasar por kernelBiseccion2. En lugar de recibir la textura de candidatos, recibe la textura de salida de la pasada anterior al método de bisección, es decir, esta información puede provenir de kernelBiseccion1 o de sí mismo. Se realizarán tantas pasadas por este kernel como sean necesarias para encontrar todos los tiempos de colisión entre los pares de objetos. El resultado ya hemos visto que será una textura de tamaño n × n, aunque para la siguiente etapa únicamente nos interesará la componente alfa que se corresponde con el tiempo de impacto.

- **Etapa de Minimización.**

  Recibe como entrada la textura de tiempos de tamaño n × n, y devuelve un único valor que se corresponde con el tiempo del primer impacto.

- **Cálculo de Impulso.**

  Si el tiempo del primer impacto es menor o igual que 1, para cada par de objetos que colisionen en ese instante de tiempo, se calculará el impulso que se debe aplicar sobre ellos; si el tiempo es mayor no se necesitará calcular los impulsos. El cálculo del impulso será llevado a cabo por un kernel y necesitará tener como entradas la posición, rotación, velocidad lineal y velocidad angular. Además también necesita que le pasemos el tiempo del primer impacto, este valor se le pasará como una variable uniforme. La forma de actuar del kernel es comprobar si los objetos correspondientes al fragmento procesado colisionan en el tiempo del primer impacto, si no colisionan no será necesario calcular impulso. En caso de que sí colisionen, el kernel avanza el estado de los objetos hasta el instante de tiempo del primer contacto y busca todos los puntos de contanto existentes entre los dos objetos que colisionan. Para todos estos puntos se calculará un único impulso total que será devuelto en el stream de impulsos. De nuevo cada fragmento procesado se corresponde a dos objetos, y por tanto en cada fragmento se devolverá el impulso resultante de la colisión entre los dos objetos correspondientes al fragmento. Por ello la textura de salida será de tamaño n × n. También vuelve a aparecer la simetría.

- **Avance de Estado.**

  Para esta etapa se usan dos kernels: uno para el caso en que existen objetos que colisionan, y otro para cuando no se encontró ningún par de objetos que colisionasen. En el primero de los casos el kernelAvanceDefinitivoConColision recibe el tiempo del primer impacto a través de una uniforme y el resto de datos, impulsos, posición, rotación, velocidad lineal y velocidad angular de los objetos, a través de texturas. Como resultado se modifican las texturas de posición, rotación, velocidad lineal y velocidad angular, con sus nuevos valores tras aplicar el impulso.

  Si el tiempo es mayor que uno, no existe ninguna colisión, y sólo habrá que modificar las rotaciones y las posiciones aplicando las velocidades correspondientes.

Como se puede ver en la figura 4.3 se usan tres streams para almacenar los datos:

- Stream de posición y velocidad lineal. Este stream es de la subclase StreamDoble, ya que se comporta como de entrada y salida en los kernels de avance definitivo. Se utiliza una textura de tipo rgba: en las dos primeras componentes se guarda la posición y en las otras dos componentes se almacena la velocidad lineal. La textura es de tamaño n×1, donde n es el número de objetos.

- Stream de rotación y velocidad angular. Este stream es de la subclase StreamDoble, ya que se comporta como entrada y salida en el kernel de avance definitivo. Se utiliza una textura de tipo rgba: en las dos primeras componentes se guarda la rotación y en las componente alfa se almacena la masa. La textura es de tamaño n×1, donde n es el número de objetos.

- Stream de índices. Es un stream de la subclase StreamSimple, ya que únicamente es de lectura. Este stream junto con el de vértices, nos permite conocer las coordenadas locales de los vértices de un determinado objeto. Es de tipo rgba: en la primera componente se guarda la coordenada del primer vértice del correspondiente polígono en la textura de vértices, en la segunda componente se almacena el número de vértices que tiene el polígono, y por último la tercera y cuarta componente guardan la masa y el momento de inercia. La textura es de tamaño n×1, donde n es el número objetos.

- Stream de vértices. Es un stream de la subclase StreamSimple, ya que únicamente es de lectura. En este stream se guardan las coordenadas de los vértices de todos los objetos, usando el sistema de referencia local del polígono. Los vértices son guardados de dos en dos en una textura de tipo rgba. En la Figura 4.5 se muestra el uso del stream de índices como una indirección sobre el stream de vértices.

  En la Figura 4.5 se puede ver cómo el objeto i tiene de masa 2 y de momento de inercia 5. También se ve que es un hexágono y que su primer vértice se encuentra en la posición 22 de la textura de vértices.

- Stream de aligned bounding boxes. Este stream es de la subclase StreamSimple. Se utiliza una textura de tipo rgba: en las dos primeras componentes se guarda el vértice inferior-izquierdo de la bounding box, y en las otras dos el vértice superior-derecho. La textura es de tamaño n×1, donde n es el número de objetos.

- Stream de candidatos. Es un stream de la subclase StreamSimple. Una vez calculadas las aligned bounding boxes se comprueba si colisionan. La textura que se utiliza es de tipo luminance, ya que sólo es necesario devolver 0 si no colisionan, y 1 en otro caso. La textura es de tamaño n×n, donde n es el número objetos. En el texel de coordenadas (i,j) se almacena la existencia o no existencia de solapamiento entre las aligned bounding boxes de los objetos i y j. En este stream vuelve a producirse

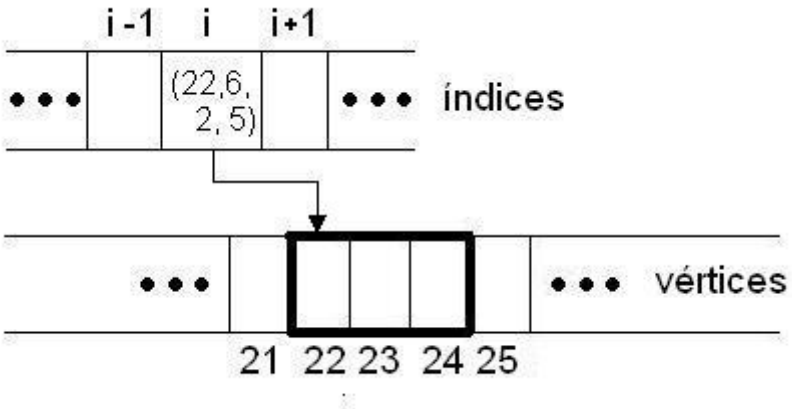


Figura 4.5: Indirección sobre el stream vértices

simetría, por lo que dentro del shader KernelAABB sólo realizaremos los cálculos para la diagonal superior de la textura.

- Stream de tiempos. Es un stream de la subclase StreamDoble. La textura que se utiliza es de tipo rgba. El tamaño de la textura es n×n, donde n es el número objetos. En la componente alfa del texel de coordenadas (i,j) se almacena el tiempo de contacto entre los objetos i y j. El resto de componentes son utilizadas para ir acotando el intervalo en las distintas pasadas por los kernels de bisección. Aquí de nuevo volvemos a usar la simetría para calcular, en los shaders de bisección, únicamente los valores correspondientes a la diagonal superior de la textura.

- Stream de impulsos. Se utiliza una textura de tipo rgba: en las dos primeras componentes se guarda el impulso que hay que aplicar a dos objetos A y B que colisionan (los correspondientes a las coordenadas de este texel), en la tercera componente el cambio en la velocidad rotacional del objeto A y en la última componente el cambio en la velocidad rotacional del objeto B. El tamaño de la textura es n×n, donde n es el número objetos. En este caso también sacamos partido de la simetría y sólo procesaremos la diagonal superior de la textura.

Capítulo 5

Resultados

5.1. Introducción

En este capítulo se presentará una comparativa entre cada una de las versiones GPU y su correspondiente versión CPU. Para realizar la comparación se han tomado tiempo de todas las etapas implicadas en la simulación variando el número de objetos de la escena. Los objetos han sido encerrados en una habitación y se les han asignado posiciones, rotaciones y velocidades iniciales iguales para las versiones CPU y GPU del mismo proyecto.

Para llevar a cabo el choque con las paredes, en el caso de la versión 2, se considera que las paredes son un objeto más, cuya masa y momento lineal es infinito, y su posición, rotación y velocidades iniciales son nulas. Para esta versión, no es necesario modificar nada de lo visto hasta ahora. Para el caso de la versión 1, el choque con las paredes se soluciona considerando las paredes como límites. Si una circunferencia se ha salido de los límites, se calcula el nuevo valor de la velocidad por reflexión. La comprobación de si un objeto se sale de los límites y la modificación de la velocidad se lleva a cabo en la etapa de avance.

Se ha intentado implementar las versiones CPU lo más parecidas posibles a las versiones GPU. Para ello, los datos que en GPU son almacenados en texturas en CPU serán almacenados en arrays. El flujo de datos de las versiones CPU es el mismo que su correspondiente versión GPU.

Los tiempos mostrados son la media de los tiempos obtenidos durante 1000 iteraciones del bucle de simulación, es decir, el tiempo total transcurrido partido por 1000. La CPU usada es un Pentium 2.8 GHz y la GPU es una NVIDIA GeForce 6200. En las dos siguientes secciones se muestran los resultados obtenidos.

Las gráficas de tiempos no deben entenderse como curvas de complejidad. Esto es así, porque la instancia para 2n puede necesitar menos tiempo que la de n. La razón está en que la configuración usada para medir los tiempos influye en el número de colisiones que

se producen. El hecho de unir los puntos de cada versión en la gráfica, los de CPU con los de CPU, y los de GPU con los de GPU, es para que sea más legible y la comparación se vea con mayor claridad.

Además las gráficas son un poco imprecisas, ya que el reloj sólo permite tomar tiempos con una precisión de milésimas de segundo.

5.2. Resultados: Versión 1

A continuación presentamos los resultados obtenidos para cada una de las etapas del proyecto. Como hemos visto las etapas del bucle de simulación para esta versión son:

- Bisección: Cáculo de normales y tiempos de contacto.
- Minimización: Búsqueda del tiempo en que se produce el primer contacto.
- Avance del estado de los objetos: Cálculo del impulso y modificación de las posiciones y velocidades de los objetos. Comprobación de si algún objeto se ha salido de la habitación y aplicar el cambio oportuno en la velocidad de dicho objeto.
- Representación gráfica: Aplicar la transformación a los vértices para su representación.

Obsérvese que hemos incluido una etapa más de las comentadas en la implementación. Esta etapa es la de representación gráfica, que se lleva a cabo por un shader de vértices. Su única función es aplicar la transformación a los vértices, es decir, los vértices se almacenan en coordenadas locales y es el shader quien se encarga de aplicarle la traslación que coloca el centro del círculo en la escena.

En la Figura 5.1 se muestran los resultados obtenidos para la etapa de Bisección. Se puede observar cómo se consigue un mejor rendimiento con la versión GPU. Esta etapa es la etapa más costosa en la versión CPU, en cambio en GPU aumentar el número de objetos influye poco en el tiempo consumido en esta etapa.

En la Figura 5.2 se muestran los resultados para la etapa de Minimización. En ambas versiones se produce un incremento del tiempo de procesado al aumentar el número de objetos, pero este aumento es mucho mayor en la versión GPU.

En la Figura 5.3 se muestra los resultados para la etapa de Avance. Con la versión GPU se consigue incrementar la velocidad en esta etapa con respecto a la versión CPU.

En la Figura 5.4 se muestra los resultados para la etapa de Representación Gráfica. En este caso la versión CPU es más eficiente que la versión GPU. En ambas, el tiempo de procesamiento aumenta según crece el número de objetos.

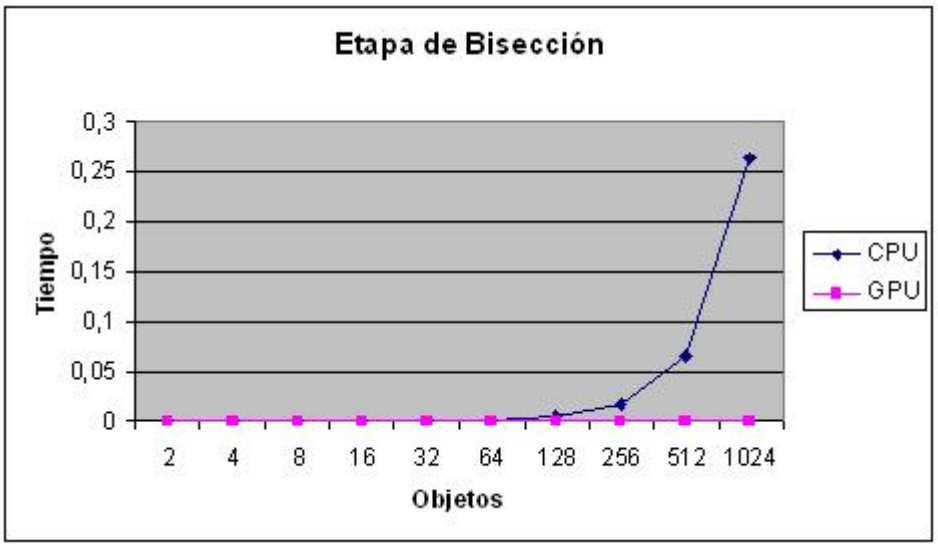


Figura 5.1: Comparativa de tiempos para la etapa de bisección (Versión 1)

Por último, en la Figura 5.5 podemos ver los resultados de tiempo para la simulación de todas las etapas. Se observa que para pocos objetos la GPU ofrece resultados peores, aunque la diferencia no es muy significativa. Según aumenta el número de objetos la versión GPU mejora su rendimiento en comparación a la CPU, y acaba superando a esta última.

5.3. Resultados: Versión 2

A continuación presentamos los resultados obtenidos para cada una de las etapas del proyecto. Como ya se ha visto las etapas del bucle de simulación para esta versión son:

- Cálculo de ABB: Se calculan las aligned bounding boxes de los objetos, al final de un paso de tiempo.
- Solapamiento de ABB: Se comprueba, por pares de objetos, si sus ABB se solapan. Los pares de objetos cuyas ABB se solapen serán candidatos a colisionar.
- Bisección: De los pares de objetos candidatos a colisionar, se comprueba si colisionan y el tiempo en que se produce el impacto.
- Minimización: Búsqueda del tiempo en que se produce el primer contacto.
- Cálculo del Impulso: Se calcula el impulso que sufren los objetos que colisionan en el mismo tiempo en que se produce el primer impacto.

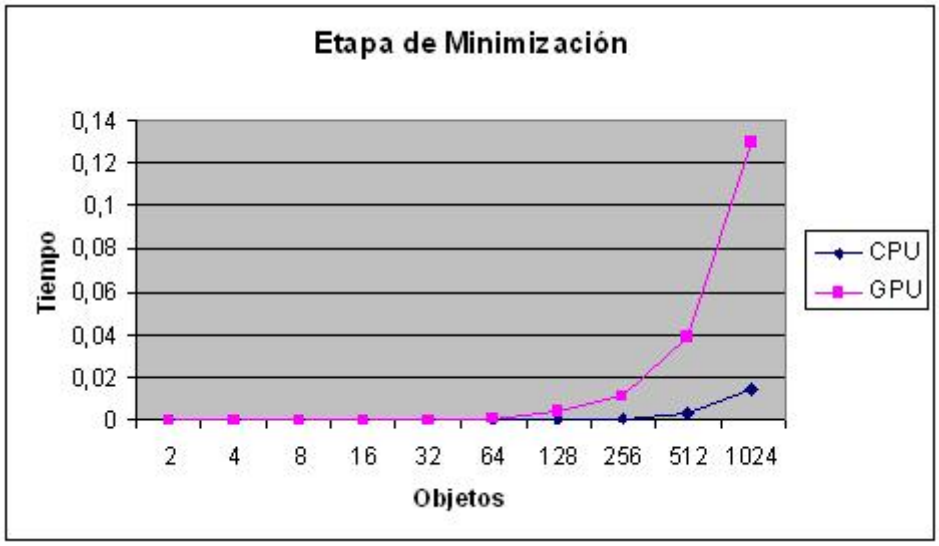


Figura 5.2: Comparativa de tiempos para la etapa de minimización (Versión 1)

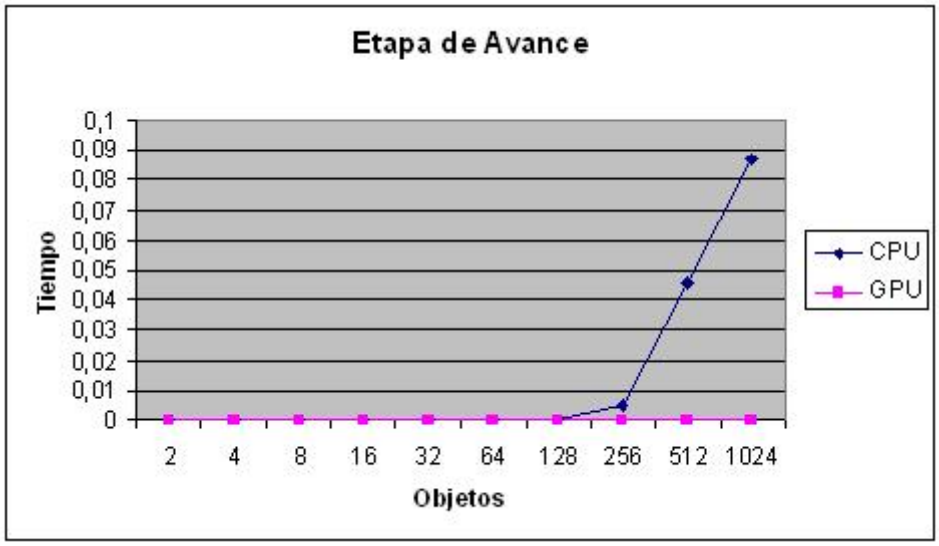


Figura 5.3: Comparativa de tiempos para la etapa de avance (Versión 1)

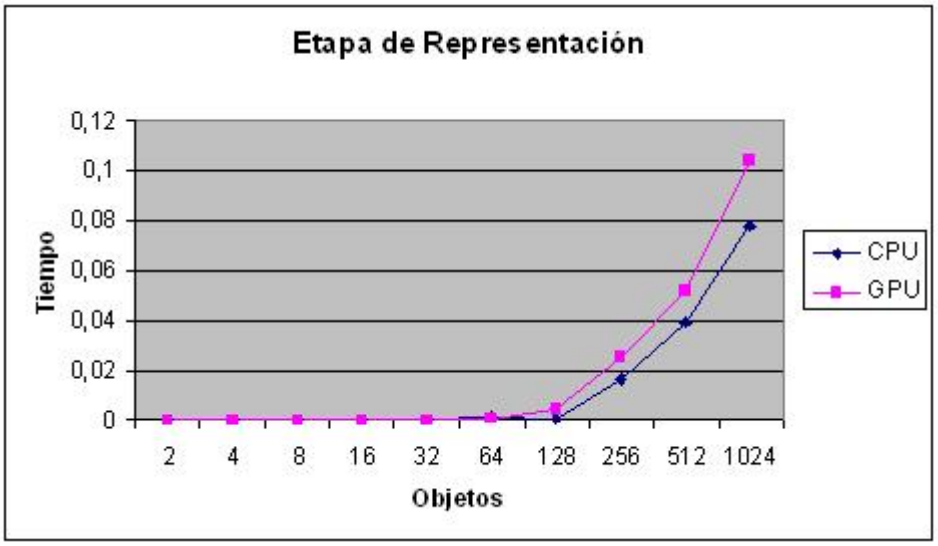


Figura 5.4: Comparativa de tiempos para la etapa de la representación gráfica (Versión 1)

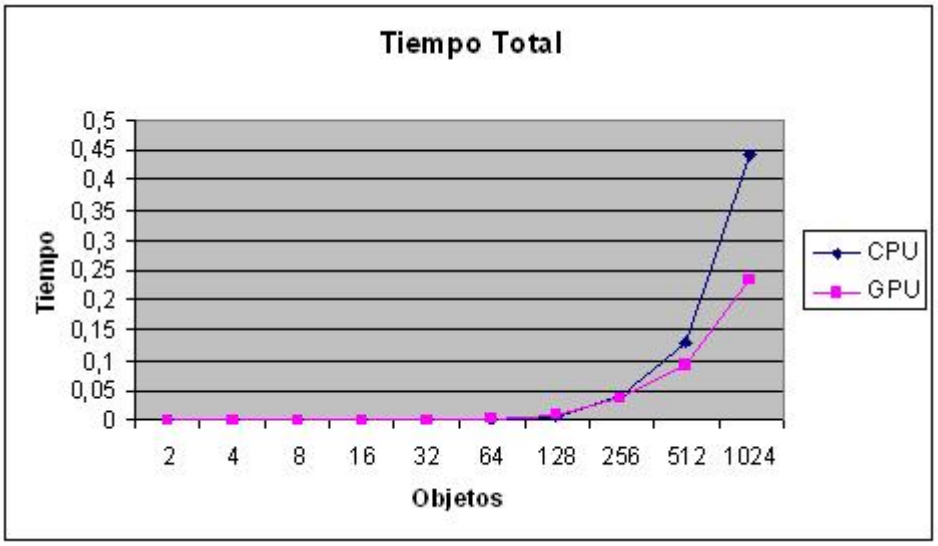


Figura 5.5: Comparativa de tiempos entre la versión cpu y gpu (Versión 1)

- Avance del estado de los Objetos: Modificación de posiciones, rotaciones y velocidades de los objetos.
- Representación Gráfica: Aplicar la transformación a los vértices para su representación.

La etapa de representación gráfica se lleva a cabo en un shader de vértices como vimos en la versión anterior, aunque ahora la transformación que se aplica a las coordenadas locales de cada vértice incluye, además de la traslación, la rotación propia del objeto.

En la Figura 5.6 se muestran los resultados obtenidos para la etapa de Cálculo de las ABB. Se puede observar que en ambas versiones el tiempo aumenta. Sin embargo la versión CPU se comporta mejor para configuraciones con pocos objetos, mientras que la GPU mejora según el número de objetos aumenta. También podemos ver que la gráfica para la versión GPU presenta una forma irregular, esto puede ser debido a la falta de precisión del reloj.

En la Figura 5.7 se muestran los resultados obtenidos para la etapa de Comprobación de Solapamiento de las ABB. En este caso se muestra superior la versión GPU. En dicha versión el tiempo apenas se modifica al aumentar el número de objetos, sin embargo en la versión CPU no sucede esto.

En la Figura 5.8 se muestran los resultados obtenidos para la etapa de Bisección. Se puede observar cómo se consigue un mejor rendimiento con la versión GPU. En ambas versiones el tiempo de procesado aumenta al crecer el número de objetos, pero este crecimiento es mucho menor en el caso de la versión GPU.

En la Figura 5.9 se muestran los resultados para la etapa de Minimización. Al igual que sucedía en la versión 1, la versión GPU se comporta peor que la CPU. Para la versión GPU esta es la etapa más costosa de todas y la que hace que el tiempo de procesado total sea peor en GPU que en CPU.

En la Figura 5.10 se muestran los resultados obtenidos para la etapa de Cálculo del Impulso. Tanto en GPU como en CPU el tiempo de esta etapa aumenta sengún crece el número de objetos, pero este aumento es mucho mayor en la versión CPU.

En la Figura 5.11 se muestra los resultados para la etapa de Avance. La versión GPU es más rápida en esta etapa.

En la Figura 5.12 se muestra los resultados para la etapa de Representación Gráfica, la versión GPU ofrece peores resultados.

Por último, en la Figura 5.13 podemos ver los resultados de tiempo para la simulación de todas las etapas. Según aumenta el número de objetos, el tiempo de procesado crece en ambas versiones. Este crecimiento es mayor en la versión GPU.

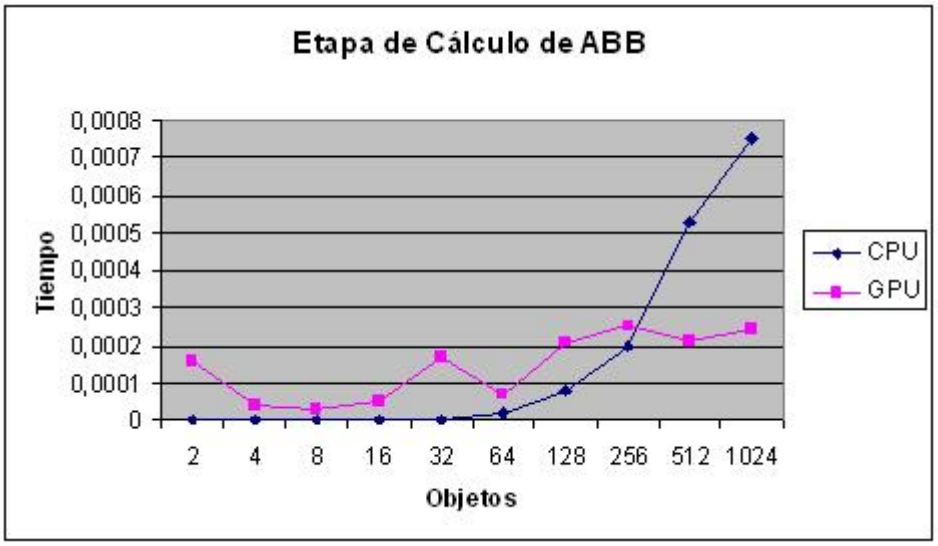


Figura 5.6: Comparativa de tiempos para la etapa de cálculo de AABB (Versión 2)

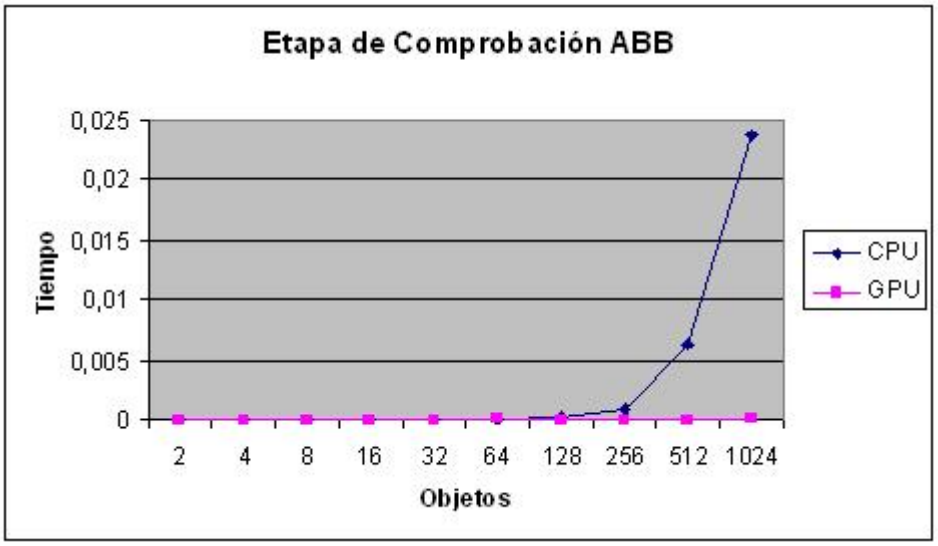


Figura 5.7: Comparativa de tiempos para la etapa de comprobación de AABB (Versión 2)

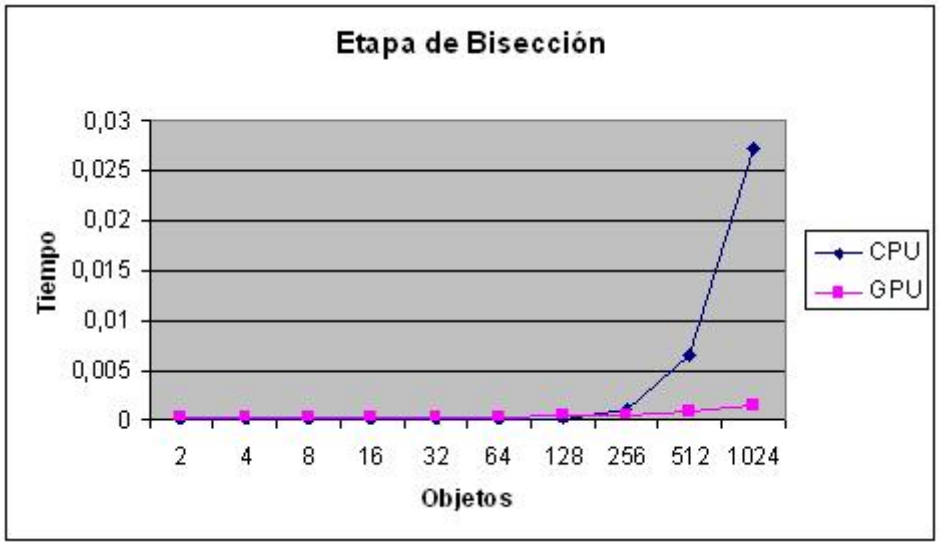


Figura 5.8: Comparativa de tiempos para la etapa de bisección (Versión 2)

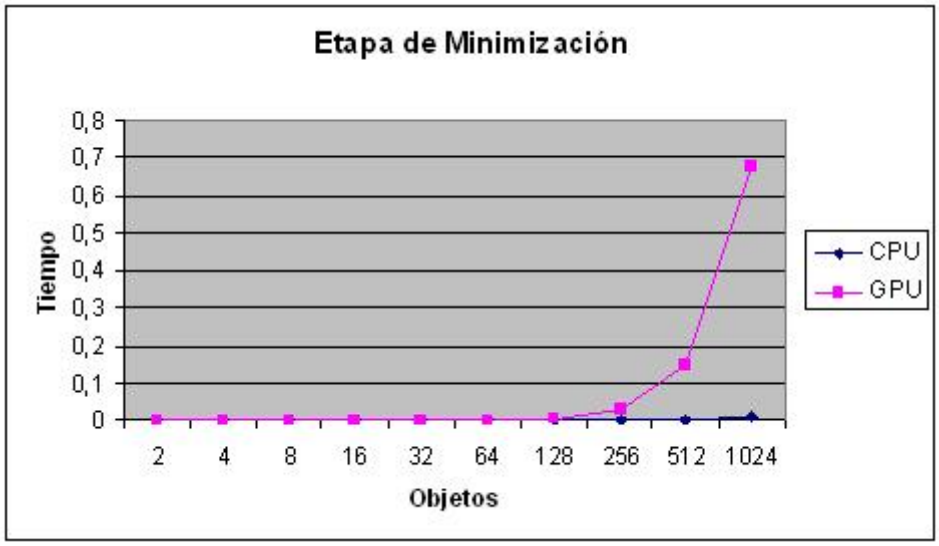


Figura 5.9: Comparativa de tiempos para la etapa de minimización (Versión 2)

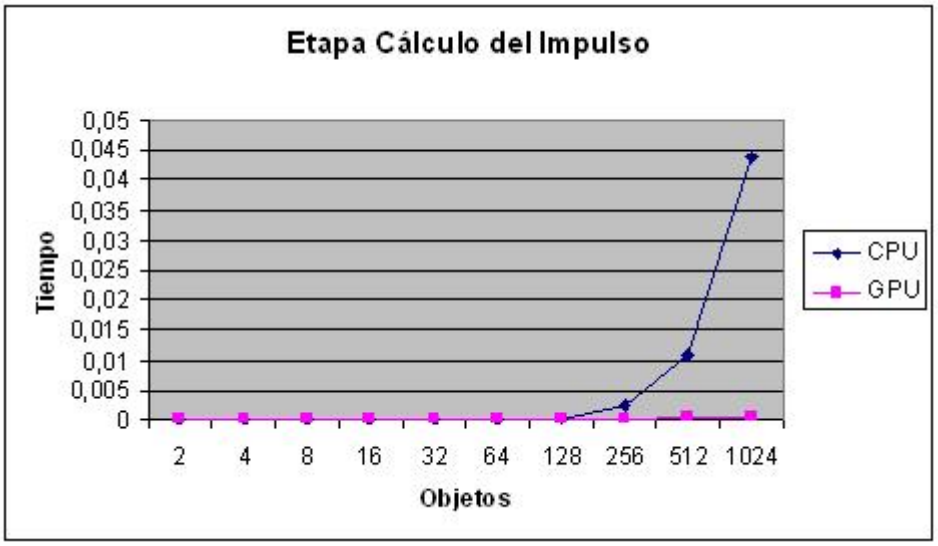


Figura 5.10: Comparativa de tiempos para la etapa de cálculo de impulsos (Versión 2)

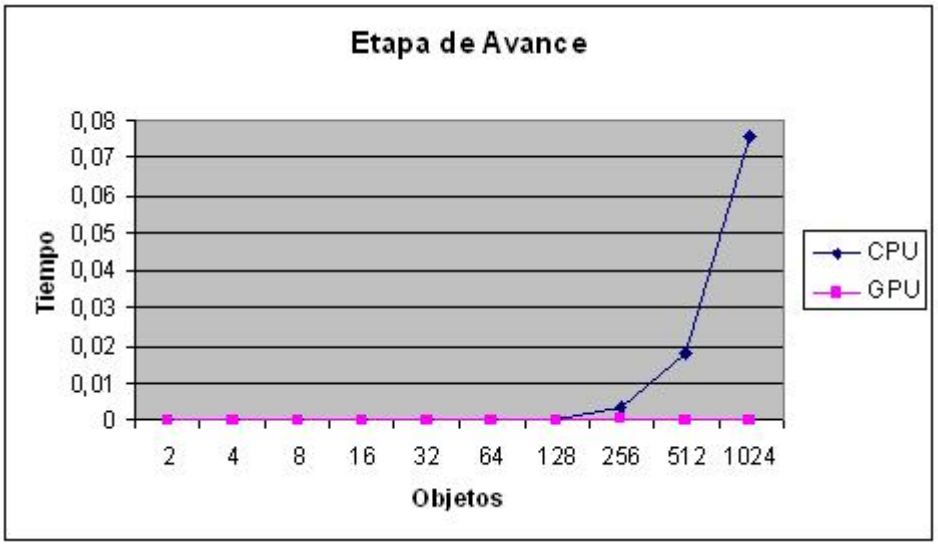


Figura 5.11: Comparativa de tiempos para la etapa de avance (Versión 2)

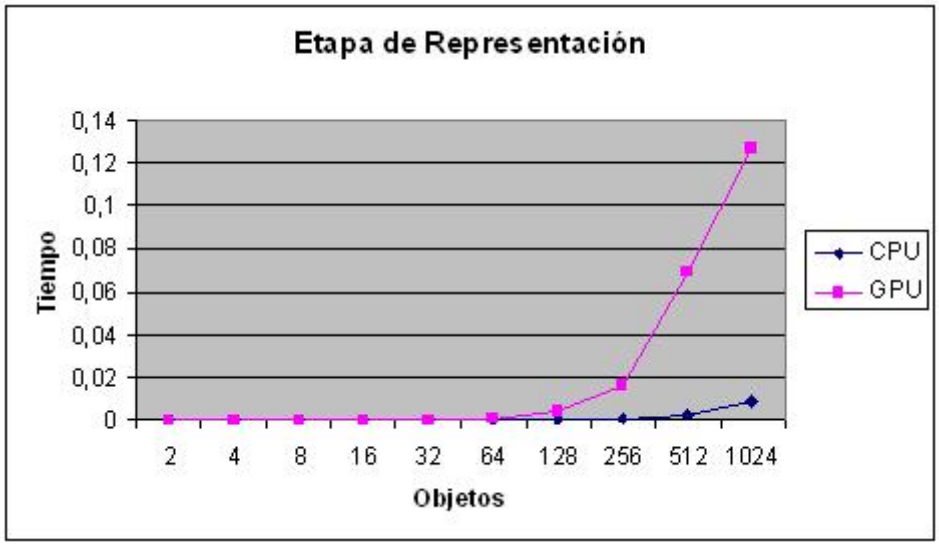


Figura 5.12: Comparativa de tiempos para la etapa de la representación gráfica (Versión 2)

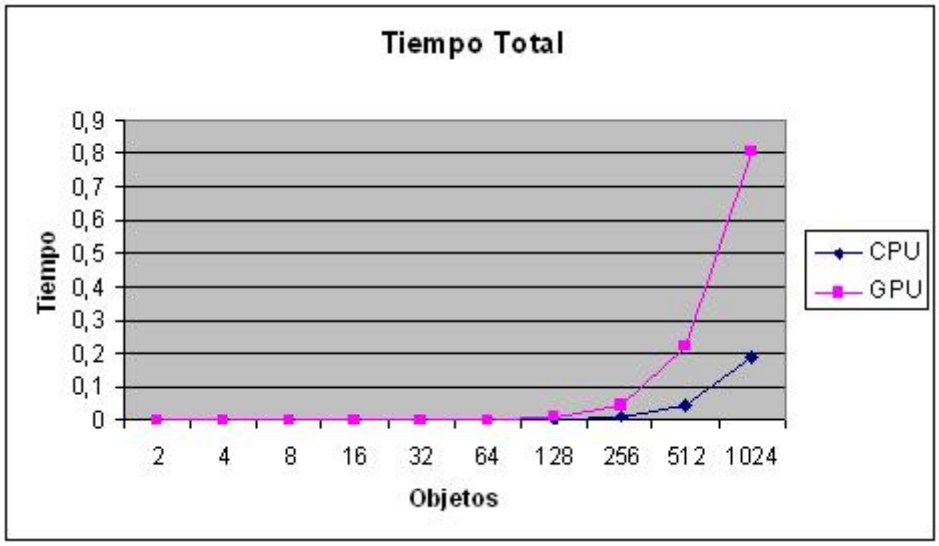


Figura 5.13: Comparativa de tiempos entre la versión cpu y gpu (Versión 2)

Capítulo 6

Conclusiones

Las tarjetas gráficas mejoran constantemente y de forma más rápida que el resto de componentes de un ordenador. Esto, junto con el hecho de que cada vez sean más programables, permite aprovecharlas para tareas que no tienen que ver con el procesado gráfico. Es aquí donde aparece el concepto de programación genérica sobre GPU, más conocido como GPGPU. La programación de GPUs permite liberar la CPU de carga de trabajo y paralelizar la ejecución de ciertas secciones de nuestro programa, consiguiendo así un incremento en el rendimiento de nuestro código. Para ello es necesario adaptar nuestro código para que pueda ejecutarse en GPU, lo cual no es sencillo. Hemos comprobado que para ciertas etapas de nuestro proyecto, aumentamos la velocidad al ejecutarlas sobre GPU, pero para otras, el resultado ha sido el contrario. Estas últimas etapas deberían ser analizadas con detalle para saber en qué punto ha fallado la implementación sobre GPU.

En general, en la mayoría de los kernels hemos conseguido aumentar su rendimiento; por ello concluimos que el uso de GPUs para la simulación de sólidos rígidos puede ofrecer buenos resultados, que serán mejores en tarjetas más potentes. Generalizando nuestra experiencia, es fácil adivinar que la programación sobre GPUs se extenderá rápidamente y aportará mejoras sustanciales sobre la programación convencional en CPU.

Apéndice A

Código

En este apéndice se va a mostrar el código de todos los shaders usados en la Versión 1. De esta forma se pretende facilitar la comprensión del funcionamiento de los shaders y del flujo de datos usados en el bucle de simulación. No se ha hecho lo mismo con la versión 2, ya que al ser más complicada no contribuye a que se entiendan mejor dichos aspectos y además supondría extender, mucho más, este documento.

A.1. Versión 1

A.1.1. Etapa de Bisección

El shader usado para esta etapa es de fragmentos. A continuación se muestra el código:

```
#version 110

uniform samplerRect texposicion;
uniform samplerRect texvelocidad;
uniform float epsilon;

float DistanciaCuadrado(vec3 punto1, vec3 punto2);

void main(){
  //Las texturas posicion y velocidad son de 1 de alto por n ancho, por
  //tanto su coordenada t siempre vale 0.5
  //Queremos rellenar la celda s,t de la matriz de colisiones. Por
  //tanto nos interesan los objetos s y t.
```

```
//La coordenada n varia entre la s y la t de la coordenada de normales
//Posicion y velocidad del objeto s
vec4 posicion1 = textureRect(texposicion,vec2(gl_TexCoord[0].s,0.5));
vec4 velocidad1 = textureRect(texvelocidad,vec2(gl_TexCoord[0].s,0.5));
//Posicion y velocidad del objeto t
vec4 posicion2 = textureRect(texposicion,vec2(gl_TexCoord[0].t,0.5));
vec4 velocidad2 = textureRect(texvelocidad,vec2(gl_TexCoord[0].t,0.5));


float tiempo=2;
vec3 normal=vec3(0,0,0);


//Rellenamos la parte de arriba de la diagonal, usaremos simetria
//Si se trata de una celda de la parte superior calculamos el tiempo
//y normal de colision entre los objetos s y t
if(gl_TexCoord[0].s>gl_TexCoord[0].t){
  float tmax=1;
  float tmin=0;
  vec3 pos1=posicion1.xyz;
  vec3 pos2=posicion2.xyz;
  vec3 vel1=velocidad1.xyz;
  vec3 vel2=velocidad2.xyz;
  float rad1=posicion1.w;
  float rad2=posicion2.w;

//Posiciones de los objetos al final de un paso de tiempo
vec3 pos1Max=pos1+vel1;
vec3 pos2Max=pos2+vel2;
//Posiciones de los objetos al comienzo
vec3 pos1Min=pos1;
vec3 pos2Min=pos2;

  //Suma de los radios
  float sumaRadiosCuadrado=(rad1+rad2-epsilon)*(rad1+rad2-epsilon);

//Si no hay interpenetracion al comienzo y la hay al final, quiere
//decir que habrá colision
if((DistanciaCuadrado(pos1Max,pos2Max)< sumaRadiosCuadrado)&&
    (DistanciaCuadrado(pos1Min,pos2Min)>= sumaRadiosCuadrado)){
    //En caso de haber colisión buscamos el tiempo de colision. Para
```

```
      //ello utilizamos bisección.
      //Mientras el intervalo de tiempo sea lo suficientemente grande
      //vamos acotandolo
      while(tmax-tmin>epsilon){
        //Se calcula el tiempo medio y las posicones en dicho tiempo
        float tmedio=(tmax+tmin)/2.0;
        vec3 pos1Medio=pos1+tmedio*vel1;
        vec3 pos2Medio=pos2+tmedio*vel2;
        //Comprobamos si hay colision el tmedio
        if( DistanciaCuadrado(pos1Medio,pos2Medio)< sumaRadiosCuadrado){
          //Si hay colision cambiamos tMax nos quedamos con el intervalo
          // en que no hay colisión y en el que si la hay
          tmax=tmedio;
        }
        else{
          //Si no hay colision cambiamos tMin nos quedamos con el
          //intervalo en que no hay colisión y en el que si la hay
          tmin=tmedio;
        }
      }
      //Una vez que el intervalo es muy pequeño, devolvemos tmin para evitar
      //interpenetración
      tiempo=tmin;
      //Sacamos la normal. La normal debe a puntar al objeto s.
      normal=vec3(posicion1.x,posicion1.y,posicion1.z)-
             vec3(posicion2.x,posicion2.y,posicion2.z);
      normal=normalize(normal);
    }
    else{
      //Sino hay colisión ponemos tiempo a 2
      tiempo=2;
      normal=vec3(0,0,0);
    }
  }
  else{
    //Si estoy en la diagonal o en la parte inferior de la matriz
    tiempo=2;
    normal=vec3(0,0,0);
  }
  //Guardamos la informacion
    gl_FragData[0]=vec4(normal,tiempo);
```

```
}


float DistanciaCuadrado(vec3 punto1, vec3 punto2){
  float aux=(punto1.x-punto2.x)*(punto1.x-punto2.x)+
            (punto1.y-punto2.y)*  (punto1.y-punto2.y)+
            (punto1.z-punto2.z)*(punto1.z-punto2.z);
  return aux;
}
```

A.1.2. Etapa de Minimización

El shader usado para esta etapa es de fragmentos. Para realizar la operación de minimización es necesario realizar varias pasadas por el shader. En la primera pasada la textura de entrada es de tipo rgba, en el resto de pasadas es luminance. A continuación se muestra el código:

```
#version 110

//Textura de entrada
uniform samplerRect texturaIn;
//Tipo de textura de entrada: 0 indica que es RGBA y 1 que es luminance
uniform int tipoTex;

void main(){
  float cx,cy;
  //Acceso a las texturas. cx y cy son las coordenadas de la textura de
  //entrada. Se multiplica por dos porque la textura de entrada es el doble
  //que la de salida.
  cx= (gl_TexCoord[0].s-0.5)*2+0.5;
  cy= (gl_TexCoord[0].t-0.5)*2+0.5;

  float minimo;

  if(tipoTex==0){
    //Si es de tipo RGBA entonces leo la componente w, que es donde
    //se encuentra el tiempo
    float t0= textureRect(texturaIn,vec2(cx,cy)).w;
    cx= cx+1.0;
    float t1= textureRect(texturaIn,vec2(cx,cy)).w;
```

```
    cy= cy+1.0;
    float t2= textureRect(texturaIn,vec2(cx,cy)).w;
    cx= cx-1.0;
    float t3= textureRect(texturaIn,vec2(cx,cy)).w;
    //cálculo del mínimo
    minimo= min(min(t0,t1),min(t2,t3));
  }
  else{
    //Si es luminance leo su unico valor
    float t0= textureRect(texturaIn,vec2(cx,cy)).x;
    cx= cx+1.0;
    float t1= textureRect(texturaIn,vec2(cx,cy)).x;
    cy= cy+1.0;
    float t2= textureRect(texturaIn,vec2(cx,cy)).x;
    cx= cx-1.0;
    float t3= textureRect(texturaIn,vec2(cx,cy)).x;
    //cálculo del mínimo
    minimo= min(min(t0,t1),min(t2,t3));
  }
  //Colocar el resultado
  gl_FragColor.x= minimo;
}
```

A.1.3. Etapa de Avance

Ya vimos que, para esta etapa se han implenetado dos shaders. Uno de ellos es usado cuando existe algun par de objetos que colisionan, el otro se usa en caso contrario. Ambos shaders son de fragmentos. A continuación se muestra el código de ambos:

- **Avance sin colisión**

```
#version 110

uniform samplerRect texposicion;
uniform samplerRect texvelocidad;
uniform samplerRect texcolision;

//Tiempo que queremos avanzar
uniform float tiempo;
//l ancho se corresponde con el numero de objetos
```

```
uniform int ancho;
//Tamaño de la habitacion
uniform float limite;

vec3 productoMatrizVector(mat3 m, vec3 v);

void main(){

  //Leemos la posicion y velocidad del objeto que corresponde
  //a este texel
  vec4 posicionA = textureRect(texposicion,gl_TexCoord[0].st);
  vec4 velocidadA = textureRect(texvelocidad,gl_TexCoord[0].st);


  //Una vez hemos recorrido todos los objetos actualizamos la
  //posición y la velocidad
  posicionA.xyz=posicionA.xyz+velocidadA.xyz*tiempo;

  //Comprobar si se sale de los límites
  if((posicionA.x>=limite)||(posicionA.x<=-limite)){
      velocidadA.x=-velocidadA.x;
      if(posicionA.x>=limite) posicionA.x=limite;
      else posicionA.x=-limite;
  }
  if(posicionA.y>=limite||posicionA.y<=-limite){
      velocidadA.y=-velocidadA.y;
      if(posicionA.y>=limite) posicionA.y=limite;
      else posicionA.y=-limite;
  }


  //Modificamos la posicion y velocidad con sus nuevos valores
  gl_FragData[0]=posicionA;
  gl_FragData[1]=velocidadA;
}
```

- **Avance con colisión**

```
#version 110
```

```
uniform samplerRect texposicion;
uniform samplerRect texvelocidad;
uniform samplerRect texcolision;

//Tiempo que queremos avanzar
uniform float tiempo;
//l ancho se corresponde con el numero de objetos
uniform int ancho;
//Tamaño de la habitacion
uniform float limite;

vec3 productoMatrizVector(mat3 m, vec3 v);

void main(){
  //Leemos la posicion y velocidad del objeto que corresponde
  //a este texel
  vec4 posicionA = textureRect(texposicion,gl_TexCoord[0].st);
  vec4 velocidadA = textureRect(texvelocidad,gl_TexCoord[0].st);

  //Queremos recorrer todos los objetos y comprobar si colisiona
  //con alguno de ellos
  float inicio=0.5;
  float fin=ancho-0.5;
  //Obtenemos la coordenada del objeto actual
  float s=gl_TexCoord[0].s;
  //En la variable incrementoVelocidad almacenaremos el cambio que
  //debemos producir en la velocidad si se producen colisiones
  vec3 incrementoVelocidad=vec3(0,0,0);

  float masaA=velocidadA.w;

  /*

  Matriz de colisiones, modo de recorrerla:

  0 0 0 1 0 0 0 0
  0 0 0 1 0 0 0 0
  0 0 0 1 0 0 0 0
  0 0 0 s 2 2 2 2
  0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

0-> Elementos que no se consultan
1->Elementos que se consultan en el primer blucle
2->Elementos que se consultan en el segundo bucle
s->Objeto actual, se encuentra en la diagonal de la matriz


*/
//Recorremos los objetos que son anteriores a s. Por tanto,
//Consultaremos los objetos de la columna s
for(float t=inicio;t<s;t=t+1){
  //Extreameos el valor de la normal y tiempo de colision
  //del objeto s con el objeto t
  vec4 colisionAB=textureRect(texcolision,vec2(s,t));
  //Leemos la posicion y velocidad del objeto t
  vec4 posicionB=textureRect(texposicion,vec2(t,0.5));
  vec4 velocidadB=textureRect(texvelocidad,vec2(t,0.5));
  //Comprobamos si el tiempo de colision se corresponde al menor
  if(colisionAB.w==tiempo){
    //En caso de ser asi, extreameos la normal
    vec3 normal=colisionAB.xyz;
    //Sacamos la velocidad relativa
    float vrel=dot(normal,(velocidadA.xyz-velocidadB.xyz));
    //Para que exista colision la velocidad relativa tiene que
    //ser menor que cero.
    //Si es mayor estamos en el caso en que se alejan y si es
    //igual en resting contact
    if(vrel<0){
      float masaB=velocidadB.w;
      float sumando1=1/masaA;
      float sumando2=1/masaB;
      //Pongo 2 en vrel porque epsilon es 1, ya que no queremos
      //que se abosrva energia
      float j=-2*vrel/(sumando1+sumando2);//+sumando3+sumando4);
      //Actualizamos el incremento de velocidad
      incrementoVelocidad=incrementoVelocidad+j*normal;
    }
  }
```

```
}

//Recorremos los objetos que son anteriores a s. Por tanto
//Consultaremos los objetos de la fila s
for(float t=s+1;t<=fin;t++){
  //Extreameos el valor de la normal y tiempo de colision
  //del objeto s con el objeto t
  vec4 colisionAB=textureRect(texcolision,vec2(t,s));
  //Leemos la posicion y velocidad del objeto t
  vec4 posicionB=textureRect(texposicion,vec2(t,0.5));
  vec4 velocidadB=textureRect(texvelocidad,vec2(t,0.5));
  //Comprobamos si el tiempo de colision se corresponde al menor
  if(colisionAB.w==tiempo){
    //En caso de ser asi, extreameos la normal
    vec3 normal=-colisionAB.xyz;
    //Sacamos la velocidad relativa
    float vrel=dot(normal,(velocidadA.xyz-velocidadB.xyz));
    //Para que exista colision la velocidad relativa tiene que
    //ser menor que cero.
    //Si es mayor estamos en el caso en que se alejan y si es
    //igual en resting contact
    if(vrel<0){
      float masaB=velocidadB.w;
      float sumando1=1/masaA;
      float sumando2=1/masaB;
      //Pongo 2 en vrel porque epsilon es 1, ya que no queremos
      //que se abosrva energia
      float j=-2*vrel/(sumando1+sumando2);//+sumando3+sumando4);
      //Actualizamos el incremento de velocidad
      incrementoVelocidad=incrementoVelocidad+j*normal;
    }
  }
}
//Una vez hemos recorrido todos los objetos actualizamos la
//posicion y la velocidad
posicionA.xyz=posicionA.xyz+velocidadA.xyz*tiempo;
//Se divide entre la masa porque el incremento es el
//incremento en movimiento lineal p=m*v
velocidadA.xyz=velocidadA.xyz+incrementoVelocidad/masaA;

//Comprobamos si el objeto se sale de los límites
```

```
    if((posicionA.x>=limite)||(posicionA.x<=-limite)){
        velocidadA.x=-velocidadA.x;
        if(posicionA.x>=limite) posicionA.x=limite;
        else posicionA.x=-limite;
    }
    if(posicionA.y>=limite||posicionA.y<=-limite){
        velocidadA.y=-velocidadA.y;
        if(posicionA.y>=limite) posicionA.y=limite;
        else posicionA.y=-limite;
    }

    //Modificamos la posición y velocidad con sus nuevos valores
    gl_FragData[0]=posicionA;
    gl_FragData[1]=velocidadA;
  }
```

A.1.4. Etapa de Representación Gráfica

El shader usado para esta etapa es de vértices. A continuación se muestra el código:

```
#version 110

uniform vec4 posicion;

void main(){
  gl_FrontColor= gl_Color;
  //Aplicar la traslación
  gl_Vertex.xyz=gl_Vertex.xyz+posicion.xyz;
  gl_TexCoord[0]= gl_MultiTexCoord0;
  gl_Position= ftransform();
}
```

Bibliografía

[Asc] Fundación Ascamm. Resistencia de materiales. http://personales.ya.com/jcmarin/download/Resistencia%20de%20materiales.pdf.

[Ast04] Dave Astle. *Beginning OpenGL Game Programming*. Course Technology PTR, first edition, March 2004.

[Ast05] Dave Astle. *More OpenGL Game Programming*. Course Technology PTR, second edition, November 2005.

[Bar97a] David Baraff. Constrained dynamics. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[Bar97b] David Baraff. Differential equation basics. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[Bar97c] David Baraff. Rigid body simulation. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[Bou01] David M. Bourg. *Physics for Game Developers*. O'Reilly, first edition, November 2001.

[Cur02] Pablo Andrés Curello. Simulación de la dinámica de cuerpos rígidos en tiempo real. Instituto Tecnológico de Buenos Aires, 2002.

[dlC] Julián Dorado de la Calle. Lenguajes de shading de alto nivel. http://sabia.tic.udc.es/gc/teoria/TrabajoHLSLs.

[Ehm99] Stephen Ehmann. Rigid body simulation tutorial. 1999.

[fac] Glsl tutorial. http://www.gpgpu.org/.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, first edition, March 2004.

[Göd] Dominik Göddeke. Gpgpu::basic math tutorial. http://www.mathematik.uni-dortmund.de/ goeddeke/gpgpu/tutorial.html.

[Hec97] Chris Hecker. Physics, Part 3: Collision response. *Game Developer Magazine*, pages 11–18, March 1997.

[Lóp] Rubén López. Programación genérica de gpu's mediante opengl. http://pinguino.dyndns.org/hospedados/gpgpu/gpgpu_ogl.html.

[Mir96] Brian Vincent Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California at Berkeley, 1996.

[Mir98] Brian Vincent Mirtich. Rigid body contact: Collision detection to force computation. Technical report, MERL - Mitshubishi Electric Research Laboratory, March 1998.

[Mir00] Brian Vincent Mirtich. Timewarp rigid body simulation. In *SIGGRAPH 00 Conference Proceedings*, July 2000.

[Pha05] Matt Pharr. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, first edition, March 2005.

[Ros04] Randi J. Rost. *Open Gl Shading Language*. Addison-Wesley Professional, first edition, March 2004.

[Vil] Jacobo Rodríguez Villar. Opengl shading language course. TyphoonLabs.