

PRÁCTICA PRINCIPAL - CREACIÓN DE UN PARQUE  
GRÁFICOS POR COMPUTADORES  
CURSO 2014 - 2015

Dayane **LIMA**

Tatiane **RODRIGUES**

Tobias **REICH**

## Contenido

### Contenido

[The final project can be found on GitHUB:](#)

<https://github.com/TobiasReich/practical4gpc.git>

### Introducción

[Estructura del proyecto](#)

[Definición de los objetos del parque \(Object definition\)](#)

[El mundo \(The World class\)](#)

[Objetos del Juego \(GameObjects\)](#)

[ASE Importer](#)

[Árboles](#)

[Incorporación de texturas \(Adding textures\)](#)

[Comparación de RGB con BGR bitmaps \(Comparision of RGB and BGR\)](#)

[Movimientos del ratón variando la vista del observador](#)

[Entrada basada de booleans \(Boolean based input\)](#)

[Vista ratón \(Mouse view\)](#)

[La detección de colisiones \(Collision detection\)](#)

[Vista multicámara \(Multi-Camera View\)](#)

[Panel de información \(HUD\)](#)

[Vista en plano cenital \(Mini map\)](#)

[Brújula \(Compass\)](#)

[SoundManager](#)

El proyecto final se puede encontrar en GitHub:

<https://github.com/TobiasReich/practical4gpc.git>

## Introducción

La *Práctica Principal de Gráficos por Computador* es, como mucho, el mayor proyecto del semestre. Con casi 4.000 líneas de código en 16 clases, hemos creado un prototipo completamente jugable de un juego en 3D.

El jugador - a través del ratón y del teclado - controla su personaje, caminando a través del paisaje misterioso de una antigua finca forestal en el medio de la noche, escuchando a la sonata de Mozart “*Claro de luna*” (“Mondscheinsonate” - Op. 27, n.º 2).

Vamos a descubrir una noche sin fin, ver el paso de la luna y escuchar los sonidos atmosféricos abajo de las estrellas.



Figura 1 - Vista del escenario del juego.

## Estructura del proyecto

Para dar al proyecto una estructura de fácil comprensión, utilizamos varias clases. Para trabajar a un nivel internacional, el código está escrito en Inglés. A partir del método principal (*main*), creamos varios objetos. Todos con sus únicos propósitos especial.

- La primera clase es el "*World*" (*mundo*), para todos los objetos dibujables. Contiene no solo todos los árboles, la casa, la tierra, sino también las luces, la luna y las estrellas. Esta clase también es responsable por la detección de colisiones con el personaje y los objetos.
- La clase virtual “*GameObject*” representa todos los objetos que se pueden visualizar. Todas las subclases tienen dos métodos: *create()* y *draw()* (*Crear* y *dibujar*). De este modo, todos los *GameObjects* pueden dibujarse a sí mismo. Así, el código permanece abierto al cambio.<sup>1</sup>
- El *ASEParser* está fuertemente relacionado con el mundo. Se analiza los archivos ASE (como la fuente) y crea objetos dibujables de estos.
- El *SoundManager* es responsable de todos los tipos de sonidos. Desde los pasos hasta la música atmosférica que se reproduce en el fondo.
- *Game* (*juego*) y *Player* (*jugador*) son clases de conveniencia. Contienen informaciones sobre el juego o el jugador como la posición del jugador o la velocidad de movimiento. Para un pequeño juego como este estas clases no son necesarios, pero proporcionan importante estructura de mantenimiento para los proyectos más grandes.

---

<sup>1</sup> Un intento de permanecer fiel al principio SOLID ([http://es.wikipedia.org/wiki/SOLID\\_%28object-oriented\\_design%29](http://es.wikipedia.org/wiki/SOLID_%28object-oriented_design%29))

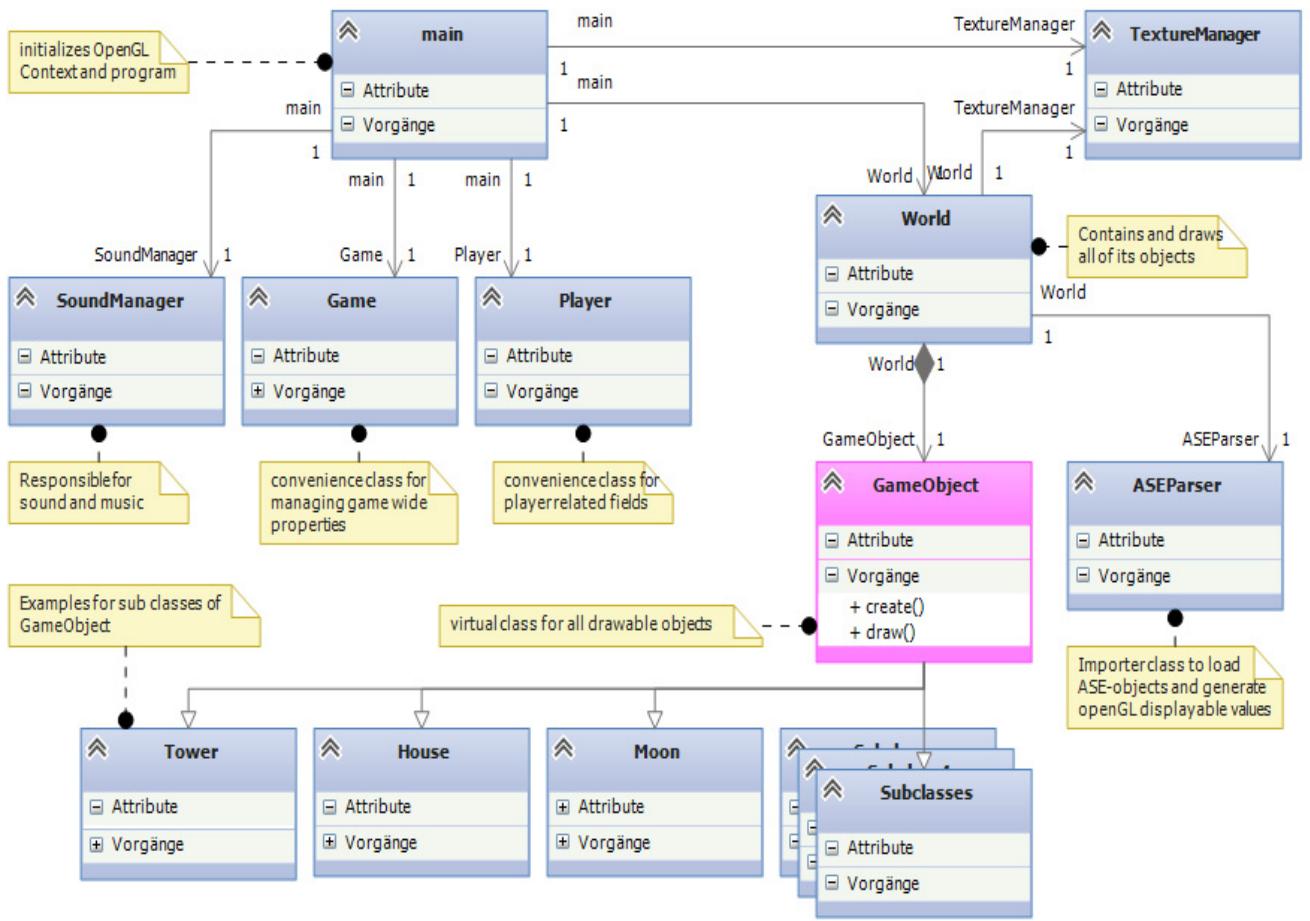


Figura 2 - Representación en clases simplificada de la estructura del proyecto.

## Definición de los objetos del parque (Object definition)

### El mundo (The *World* class)

La clase "*World*" es, literalmente el centro del juego. Una clase que contiene todos los objetos dibujables como los árboles, la casa, la luna y las estrellas.

Para más comodidad y mejor rendimiento, básicamente todos estos objetos se almacenan en las listas de visualización (*display lists*). Estos - una vez creado y compilado - almacenan los objetos directamente en la GPU. Esto da al proceso de dibujo un enorme aumento de rendimiento.<sup>2</sup>

Cada vez que se actualice la pantalla, lo único que tiene que hacer es definir la matriz de transformación y llamar a la lista de visualización en la GPU.

### Objetos del Juego (*GameObjects*)

La clase de *GameObject* es una clase abstracta. Contiene dos métodos importantes:

- *create* (crear);
- *draw* (dibujar).

Cada objeto dibujable en la escena *3D* es una subclase de *GameObject* y por lo tanto hereda esos métodos. De esta manera la clase *World* sólo es responsable de la gestión de *GameObjects* dibujables, así no necesita haber en su código comandos relacionados a *OpenGL* de los objetos.

Subclases conocidos son:

- *House*
- *Lake*
- *Moon*
- *Skyline / Clouds*

<sup>2</sup> Más: <https://www.opengl.org/archives/resources/faq/technical/displaylist.htm>

- *Starfield*
- *Tower*
- *Ground*
- *Trees*
- *Fountain*

### ***ASE Importer***

Algunos de los objetos fueran criados directamente en el código, pero otros son importados de fuentes externas. Sobre la base de la *ASE-Importer* de Practica 3 importamos por ejemplo el archivo: *fuente.ase*

El importador se cambió ligeramente, ya que en la Practica 3 utilizamos la idea de crear un objetos de superficies3D con 3 índices de *vértice* y un *normal*. Ahora utilizamos normales de cada *vertex* para obtener un sombreado suave. Usando esta técnica, las normales entre los vértices normales son interpolados - lo que conduce a un sombreado más fino.<sup>3</sup>

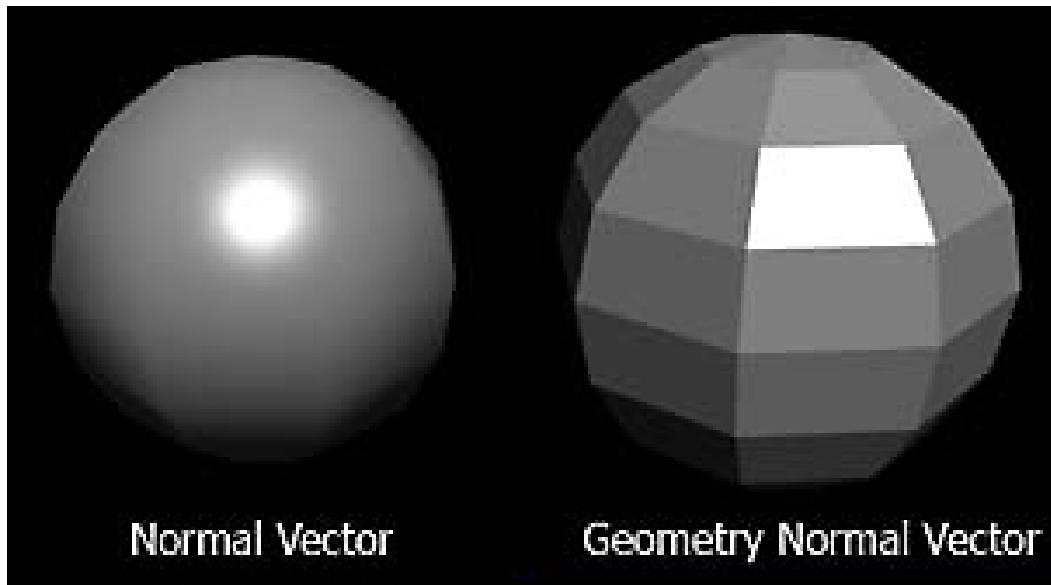


Figura 3 - Comparación de las normales por cada vertex y un normal por superficie.

<sup>3</sup> Más sobre esto (y la fuente de este dibujo) aquí:

<http://gamedev.stackexchange.com/questions/50653/opengl-why-do-i-have-to-set-a-normal-with-glnormal>

## Árboles

Una parte del proyecto era la creación de diferentes tipos de árboles para el parque. Dado que nuestro objetivo era más en aprender OpenGL que 3D Studio Max, hemos decidido crear los árboles en lugar de una simple importación de modelos 3D. Esto sería simplemente otra llamada para el importador ASE - sin ninguna dificultad.

En lugar de eso, decidimos utilizar objetos OpenGL basado en estilos 3D para crear objetos más complejos que el suelo o la torre. El siguiente gráfico muestra los tres tipos diferentes de árboles y la forma en que fueran creados, desde objetos geométricos básicos. Todos ellos son creados y dibujados por otra subclase de `GameObject`, los árboles. Esta clase ofrece tres métodos para dibujarlos.

1. `void drawSmallTree (void);`
2. `void drawMediumTree (void);`
3. `void drawBigTree (void);`

El mundo (`World`) almacena una lista de posiciones para cada uno de ellos y los utiliza - como veremos más adelante - para la detección de colisiones.

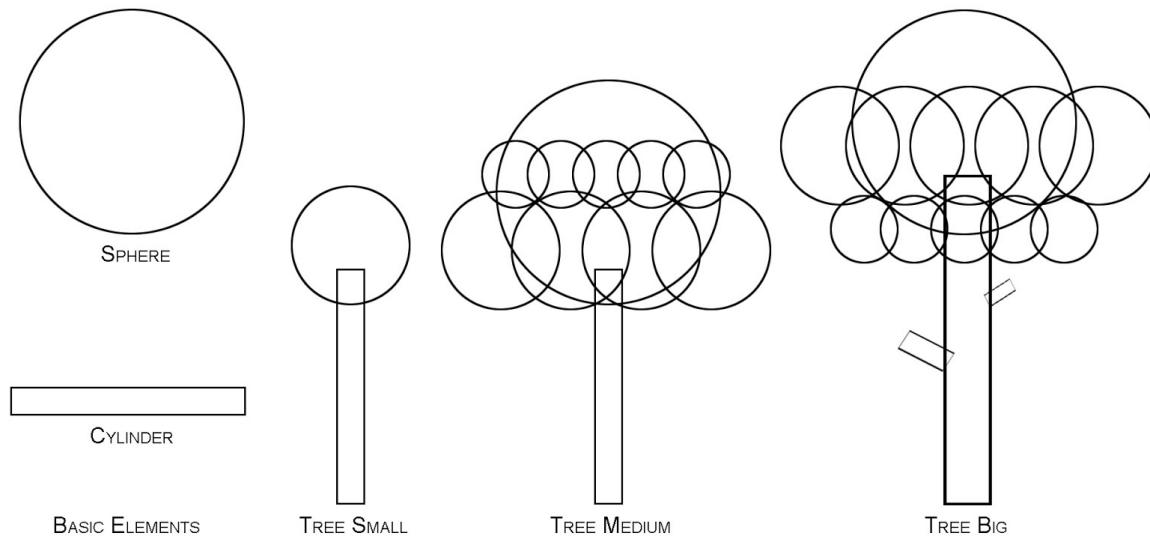


Figura 4 - Descripción esquemática de la construcción del árbol

## Incorporación de texturas (Adding textures)

La idea inicial era escribir un método capaz de leer los *bitmaps* del nuestro archivo de textura. Después, copiamos los datos al almacenamiento *OpenGL Texture2D*. La ventaja de esta función es la menor dependencia de otros marcos (*frameworks*), como la biblioteca *DevIL*.

Por supuesto, esto tiene un inconveniente. Por ejemplo, sólo los archivos de bitmaps puros podían ser cargados y utilizados, pero no los archivos de *jpg* o *png*. El método está disponible (aunque *deprecated*) pero reemplazado por el de *DevIL*.

### **Comparación de RGB con BGR bitmaps (Comparision of RGB and BGR)**

Otro gran inconveniente para la propia función de importación de archivos de textura es el formato de archivo. Uno se utiliza para programas gráficos y generalmente utilizan el formato RGB<sup>4</sup> (*Red Green Blue*). Lamentablemente, por razones de rendimiento de datos gráficos se almacenan (internamente) en formato BGR (*Blue Green Red*) en el *hardware*. Esto conduce a valores de color conmutadas de rojo y azul.

La siguiente imagen muestra el resultado:

---

<sup>4</sup> Hay varios tipos, como RGBA (Alpha) o sólo RGB en diferente profundidad de bits (R5 G6 B5, R8 G8 B8, A1 R5 G5 B5, ...). Para el alivio sólo, declaramos todos RGB.



Figura 5 - Ejemplo de resultados equivocados con almacenamientos diferente de valores RGB.

Por esto las versiones de *OpenGL* superior a 1.1 ofertan no sólo *GL\_RGB* pero también el formato *GL\_BGR*. de la función *glTexImage2D()*. Por razones de herencia, decidimos mantener la versión que sea, evitando profundizar el cambio de las bibliotecas importadas.

La solución simple para esto fue usar *DevIL*. Este biblioteca de imágenes ofrece una amplia cantidad de funcionalidad. Entre que una función para importar texturas. La siguiente línea reemplaza la función escrita por nosotros. *DevIL* requieres el nombre del archivo para cargar y devuelve el *Integer-ID* de la textura de *OpenGL*.

```
texture = ilutGLLoadImage (const_cast<char*> (filename));
```

Nuestra función utiliza además un booleano para determinar si la textura se repetirá o no. El valor predeterminado es repetir la textura por lo que en caso de que no se quiere que utilizamos la siguiente línea:

```
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Esto es útil ya que algunas texturas se repiten a si mismo (la valla, paredes, suelo), mientras que otros tienen que sujetarse (la brújula, elementos del *HUD* ...). La función `initTextures (void)` utiliza el método antes mencionado para inicializar las 27 texturas del juego.

## Movimientos del ratón variando la vista del observador

### Entrada basada de booleans (*Boolean based input*)

Basado en los proyectos anteriores, hemos optimizado el sistema de entrada del teclado y del ratón. El cambio más importante fue hacer un sistema más flexible, basado en valores booleanos para cada tecla pulsada. Esto da más flexibilidad para varios insumos simultáneos.

La primera idea fue la de esperar a una tecla pulsada y reaccionar, pero con esa idea vinieron algunos problemas: Si el jugador está presionando más de una tecla, la alarma (interrupt) desde el teclado sólo señalará la última tecla presionada. Si el jugador desea avanzar pero esquivar a un lado al mismo tiempo, el teclado sólo reaccionaría a la última tecla.

El sistema nuevo utiliza básicamente 3 funciones:

- *KeyDown-Listener*
- *KeyUp-Listener*
- *Keyboard-processor*

Mientras los KeyDown (tecla abajo) / KeyUp (arriba) - Listeners básicamente sólo se establecen el booleano relacionado con la tecla (empujar = `true` / dejarlo = `false`) el Keyboard-processor hace todo el trabajo. Éste está preguntando para todos los valores del teclado. Por ejemplo, si el booleano `wPressed` esta `true`, se ejecuta la acción de esta clave "W" (avanzar). Esto permite, básicamente, una cantidad ilimitada de teclas se pulsen sin preocuparse acerca de un problema de interrupción de teclado.

### Vista ratón (*Mouse view*)

La entrada para la vista del ratón funciona de una manera ligeramente diferente. Para cada fotograma (*frame*) el programa comprueba la posición del cursor y la compara con la posición de la última fotograma. La diferencia se transforma en una rotación de la cámara carácter. Si el cursor del ratón llega a un área cerca del borde de la ventana, será "*warped*" a una posición central. Así el cursor nunca puede ser "detenido" en el borde de la pantalla.

## La detección de colisiones (*Collision detection*)

El jugador puede circular libremente dentro de la escena. Por eso, puede caminar para direcciones que, por ejemplo, no desea o que no sea la correcta. Por esa razón una detección de colisiones es necesaria.

En los juegos, por lo general, hay dos ideas de detección de colisiones:

- Colisión por la Superficie (*Per-Surface Collision*): muy lento, pero precisa;
- Cuadros Delimitadores (*Bounding Boxes*): muy rápido, precisión depende del objeto.

Por motivos de rendimiento, sino también a la falta de tiempo - la implementación de una completa detección de colisiones por superficie para todos los objetos no es ciertamente una tarea para un solo semestre - utilizamos la técnica de cuadros delimitadores (*Bounding Boxes*).

En esta técnica cada objeto tiene otra versión simplificada del mismo objeto como un cuadro de límite. Por lo general, esto es una (o mas) caja(s) o esfera(s). Ambos tipos hacen que sea fácil de calcular si un punto está cerca (o dentro del objeto) o no.

Mientras que la esfera tiene que hacer cálculos más complejos para la distancia (Pitágoras) la caja sólo tiene que comprobar si el punto está a la izquierda o derecha / antes-detrás del centro de la caja. El inconveniente es, objetos circulares no pueden exactamente ser representados con cajas mientras que los objetos rectangulares no pueden describirse fácilmente con círculos.

Hemos simplificado este cálculo más, chequeando solo en 2 dimensiones (por ejemplo un jugador nunca subir a un árbol). El siguiente gráfico muestra los cuadros delimitadores (caja y esfera) para ambos árboles y la casa:

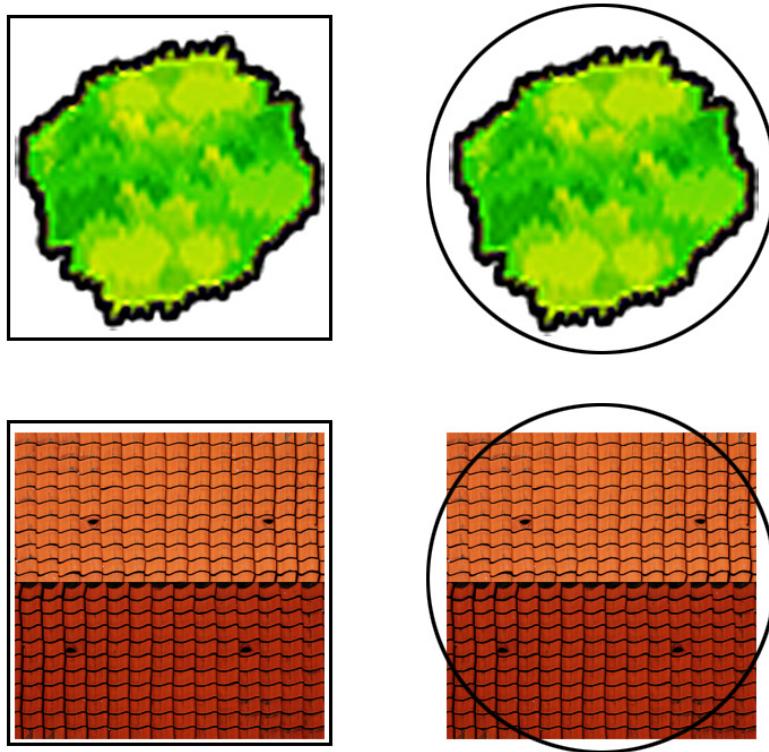


Figura 6 - Ejemplo para cuadros delimitadores de los árboles y la casa.

Decidimos utilizar la versión en caja. Para eso, la clase *World*, que se mantiene un registro de todos los objetos, tiene el método:

```
bool World::checkCollision(float newX, float newZ)
```

El punto de X y Z dado se cotejará con todos los árboles y otros objetos. Así también las coordenadas del mundo (por lo que el jugador no puede salir). El valor de retorno indica si hubo una colisión o no. Así, volvemos otra vez en el *Keyboard-processor* (procesador del teclado). La lógica es la siguiente:

```
if (aPressed) { // STRAFE LEFT

    newX = player.eyeX + Player::WALKING_SPEED
        * cos (player.rotLeftRight * PI / 180.0);

    newZ = player.eyeZ - Player::WALKING_SPEED
        * sin (player.rotLeftRight * PI / 180.0);

    if (! world.checkCollision(newX, newZ)){ // collision test
        soundManager.isMoving = true;
        player.eyeX = newX;
        player.eyeZ = newZ;
    } else {
        soundManager.playCharacterHitSound();
    }
}
```

Si la nueva posición del jugador ofrece ninguna colisión, nos movemos a ese lugar, y tenemos la reproducción del sonido movimiento. De lo contrario, tiene que haber una colisión y que activar el efecto de sonido relacionados (*SoundManager*).

## Vista multicámara (*Multi-Camera View*)

Hay varias cámaras en la escena. No sólo la del jugador. Usando el botón del teclado “C” el jugador puede cambiar a través de todos ellos.



Figura 7 - Vista de la cámara 2 (Sur Este).

El método void `setCameraPosition()` es responsable por elegir la camera adecuada. Llevamos un registro del numero entero para la posición de la cámara, de acuerdo con el valor actual de este método utiliza `gluLookAt()` para establecer la cámara correcta.

Una de las tareas era que todas las cámaras están moviendo - independientemente de si el jugador está viendo la imagen de ellos o no. Para evitar cálculos permanentes para todas las cinco cámaras "automáticas", su movimiento y al centro, la nueva solución era simplemente hacer esto una vez por todas.

El método antes mencionado requiere el cálculo `gluLookAt()` sólo para el seleccionado actualmente. Pero hemos elegido vincular todos los movimientos juntos para un mejor rendimiento.

Para resolver eso creamos un "*punto de enfoque*" en la escena. Esto es alrededor del centro de la escena (y la casa). Este punto de enfoque es el destino de la posición `atX`, `atY`, `atZ` para cada una de estas cámaras. En lugar de calcular los cinco posiciones de cámara y la rotación de cada fotograma, sólo tenemos que calcular este punto cada fotograma.

La siguiente función hace la actualización:

```
void World::updateCameras (void) {
    camRotation += 0.2f;
    camDest.lat = 15 * sin(camRotation * PI / 180.0);
    camDest.lon = 15 * cos(camRotation * PI / 180.0);
}
```

El punto de enfoque gira alrededor del centro en un radio de **15** unidades (metros). La cámara seleccionada actualmente utiliza este punto por su centro. De esta manera todas las cámaras están moviendo permanentemente, sin el cálculo de los puntos centrales de cada uno de una manera diferente. Incluso la cámara casa puede utilizar esta función para el círculo alrededor de sí mismo.

La siguiente imagen ilustra la idea:

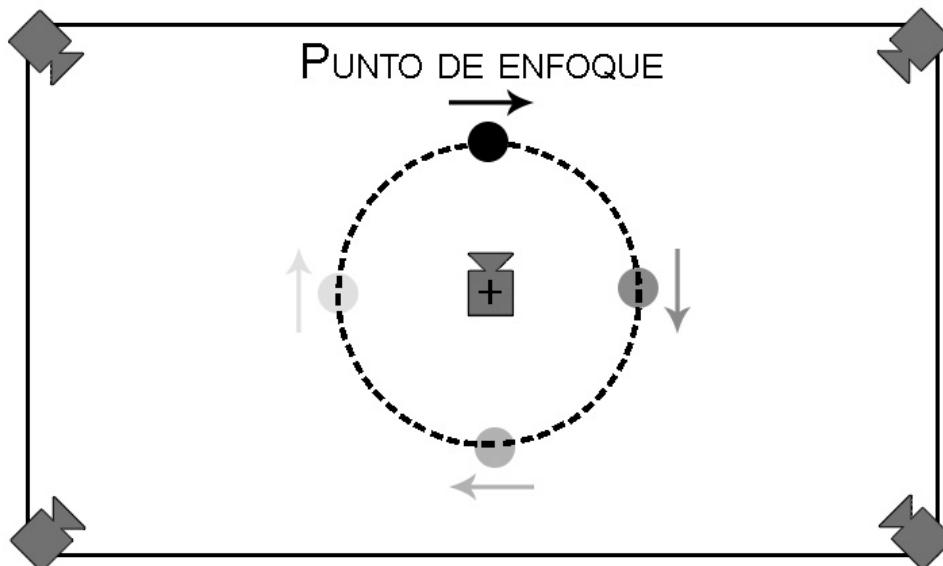


Figura 8 - Ilustración del punto de enfoque común utilizado para todas las cámaras.

Para mostrar al jugador mientras que mirando a través de una de las cámaras de vigilancia, se utiliza la posición y orientación del jugador para hacer un modelo de "carácter". Esto solo pasa mientras que el *ID* de la cámara no es uno de los jugadores.

Usando esta técnica podemos ver y mover el jugador mientras se ve la escena a través de una de las otras cámaras.



Figura 9 - Ejemplo del jugador de verse a sí mismo a través de las cámaras.

El HUD, el mapa y la brújula ofrecen ayuda adicional para el movimiento y mantener siempre fijos en la posición y orientación del jugador.

## Panel de información (HUD)

El *HUD* (*Heads Up Display*) proporciona toda la información esencial para el jugador. La información se agrupa en las siguientes partes:

- Un **mapa** (*mini map*) con todos los objetos en el mundo (jugador, casa, árboles, ...)
- Una **brújula** (*compass*) con la orientación del jugador
- El **número de la cámara** (en caso de que no son los jugadores ver)
- **Ayuda en línea** (*online help*) con todas las claves disponibles
- **Estadísticas**, fps (fotogramas por segundo)
- El **punto de mira** en el centro (*crosshair*)
- **Información** sobre la luz, sonidos y música

Básicamente, todos los elementos del *HUD* funcionan de la misma manera. Después de dibujar la escena cambiamos a un modo *OpenGL 2D* ortográfica para dibujar los elementos. Considerando que el fps y el “punto de mira” siempre están pintados, los demás elementos se pueden activar y desactivar.

El método `drawHUD` se ve así:

```
void drawHUD(void) {
    setOrthographicProjection();
    drawCrosshair();
    drawFPS();
    if (showInfo)
        drawStats();
    if (showCompass)
        drawCompass();
    if (showMinimap)
        drawMinimap();
    if (camNumber != Game::CAM_PLAYER)
        drawCamLogo();
    resetPerspectiveProjection();
}
```

## Vista en plano cenital (*Mini map*)

Hemos decidido utilizando el botón "P" del teclado no sólo para mostrar un plano cenital en toda la pantalla, sino también a mantener el resto de la escena visible.

En la parte superior de la escena 3D estamos proporcionando una visión general del mundo - el llamado “*mini map*”. De esta manera el jugador tiene la libertad de ver el mapa 2D del mundo y el espacio 3D al mismo tiempo. Esto le permite usar tanto la información al mismo tiempo.

El “*mini map*” es el elemento más complejo de todos los elementos 2D. Consta no sólo de una capa de fondo semi-transparente, sino también de todos los objetos que se dibujan en la parte superior de la misma.



Figura 10 - El fondo del “*mini map*”.

Todos los árboles, el lago y la fuente, ytambién el símbolo del jugador se dibujan en la parte superior del fondo. Además dos símbolos en la parte superior derecha del minimapa indican la luz del jugador y / o la luz de la luna estan activado (*a color*) o desactivado (*gris*).

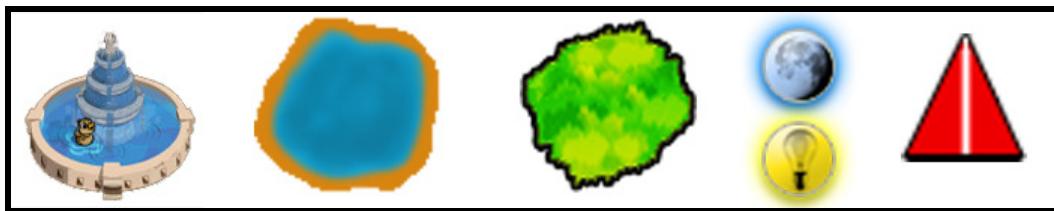


Figura 11 - Texturas de objetos dinamicos (la fuente, el lago, arboles, las luces, el jugador).

El “mini map” final se ve como en la siguiente imagen. Aquí el jugador se coloca cerca de un árbol entre el lago y la casa. La luz de la luna y la luz del jugador están encendidos.



Figura 12 - El “mini map” final.

## Brújula (Compass)

La brújula (parte arriba-derecha de la pantalla) muestra la orientación del jugador. Se mantiene siempre visible, incluso si el jugador elige otra cámara. Sin embargo, se puede desactivar a través del teclado (la tecla “K”).



Figura 13 - La brújula y sus partes en segundo plano<sup>5</sup>.

<sup>5</sup> El fondo se compone de dos capas separadas. Esto permite invertir la brújula (la rotación de la escala y no la aguja si se desea), de modo que la aguja sigue apuntando hacia el norte y el dial brújula es la rotación en la dirección opuesta.

## SoundManager

La clase `SoundManager` es responsable de la representación acústica del juego. Esa clase incluye pasos, sonidos de colisión (con árboles, la cerca, la casa, etc.), y la música de fondo también. La funcionalidad es simple. Justo al comenzar, el `SoundManager` carga todos los sonidos y la música. Esto se hace en:

```
void SoundManager::initSound (void)
```

Aquí todos los sonidos se cargan. Justo después de la carga de la música, el `SoundManager` comienza a reproducirlo. Las pisadas sin embargo se quedan en pausa.

Conectado con el movimiento invocamos el `SoundManager`. La forma más fácil es:

```
if (! world.checkCollision(newX, newZ)) {
    soundManager.isMoving = true;
    player.eyeX = newX;
    player.eyeZ = newZ;
} else {
    soundManager.playCharacterHitSound();
}
```

Toda vez que el jugador deje de moverse, el `SoundManager` es notificado y juega un sonido de colisión (*hit sound*).

De la misma manera el `SoundManager` se invoca cada cuadro para reproducir (o no) un sonido del movimiento. El `SoundManager` está comprobando si el jugador está en movimiento. En caso afirmativo y con los sonidos activados, otra vez va a ser comprobado si se detienen los ruidos de pisadas. Si es así, los ruidos se reanudarán. De esta manera, los sonidos están conectados directamente al movimiento del jugador.

```
void SoundManager::playCharacterFootsteps (void) {
    if (isMoving && playSound) {
        if (Mix_Paused(soundChannelFootSteps))
            Mix_Resume (soundChannelFootSteps);
    } else
        Mix_Pause (soundChannelFootSteps);
}
```