

Universidad Tecnológica Nacional

Facultad Regional Buenos Aires

Técnicas de Gráficos por Computadora

Trabajo Práctico: “Commandos”



Grupo ValePorUnNombreGeek

- Pablo Santiago Fernández (141.266-8)
- Matías García Isaía (134.436-5)
- Daniela Kazarian (140.873-2)

Introducción

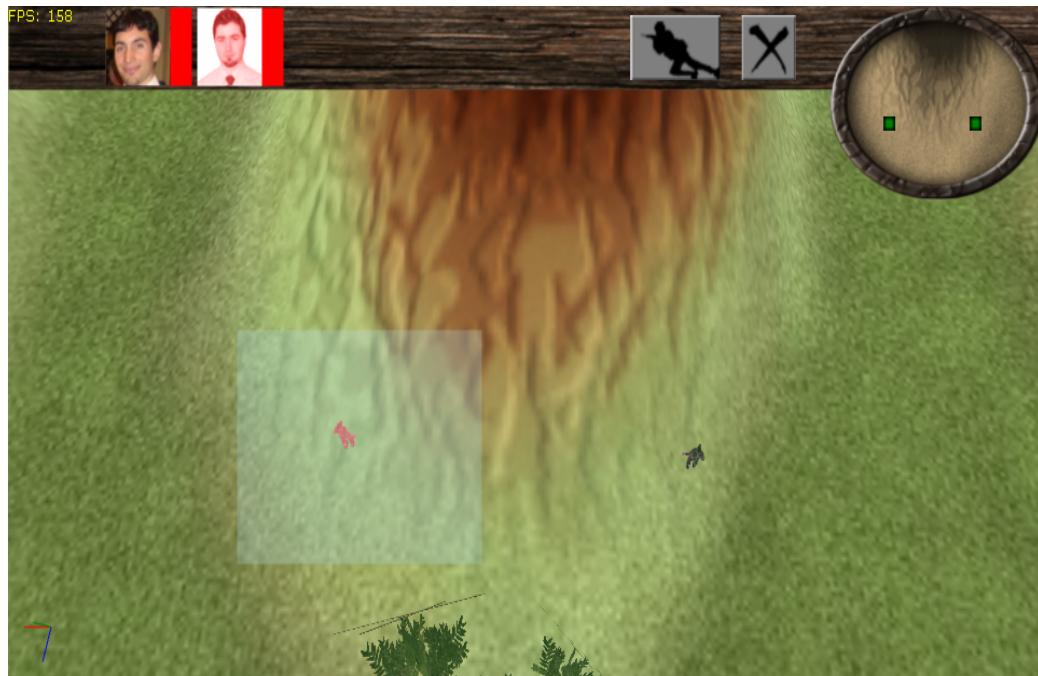
El trabajo práctico consistirá en implementar un clon del juego Commandos, pero con gráficos en tres dimensiones.



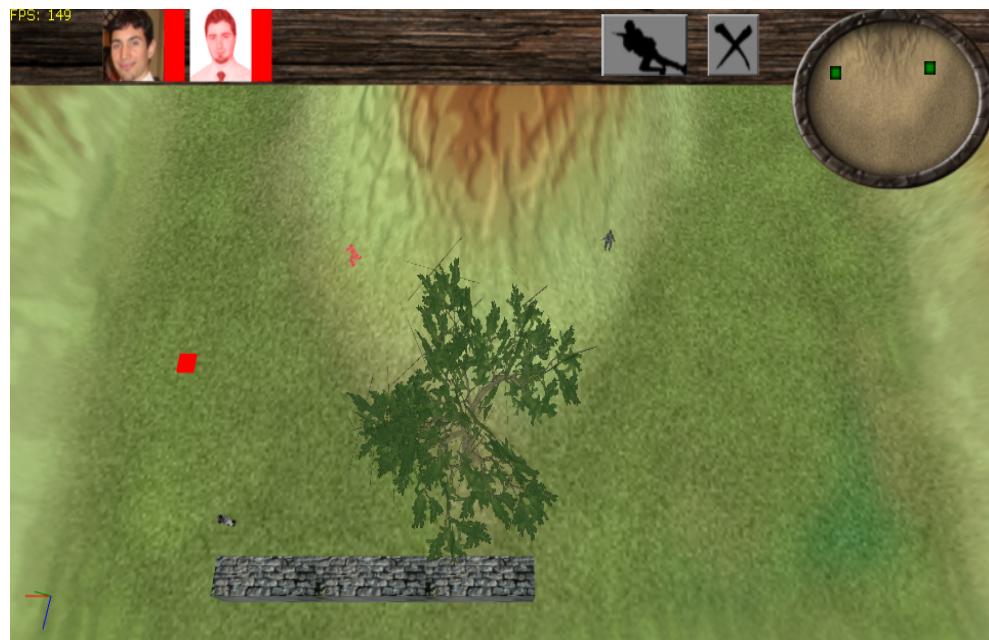
Commandos es un juego de estrategia en tiempo real en el que el jugador controla un grupo reducido de comandos de guerra a través de distintos escenarios. En cada uno de estos, el jugador deberá guiar a su tropa para eliminar a los soldados nazis y cumplir distintos objetivos evitando ser descubiertos por el enemigo.

Modo de juego

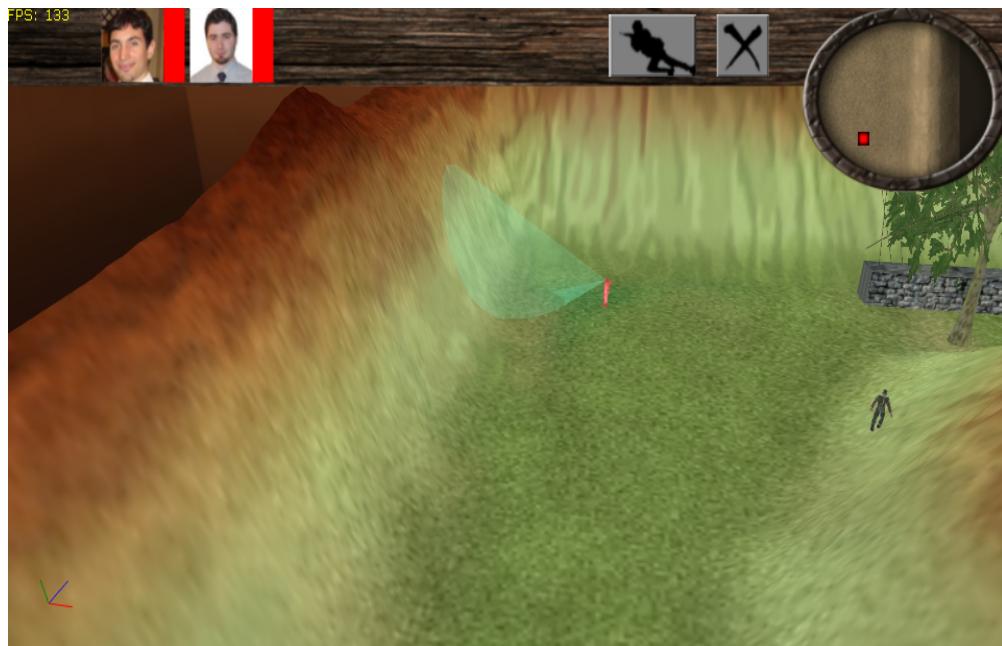
El usuario puede seleccionar a sus personajes tanto clickeando su foto como su modelo, o bien seleccionando un área rectangular en pantalla con el mouse.



Con click derecho se le ordena a los personajes activos el destino al cual dirigirse. En caso de que el destino sea un soldado enemigo, el personaje lo perseguirá y matará cuando esté próximo al mismo.



Seleccionando un soldado enemigo puede verse su cono de visión, el volumen dentro del cual será capaz de detectar y atacar a nuestros Comandos.



Usando las flechas del teclado o acercando el mouse a la mitad de cada uno de los bordes de la pantalla de juego se puede desplazar la cámara por el terreno. Además, con el scroll del mouse o con las teclas A y Z se puede modificar el zoom.



Por último, presionando la rueda del mouse y moviéndolo se puede rotar la cámara de visión.

Detalles de implementación

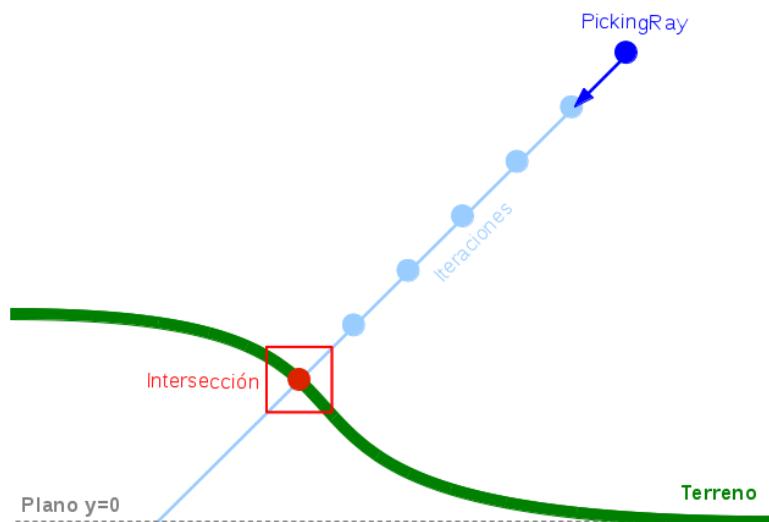
A continuación destacamos las características más importantes que desarrollamos y aplican conceptos de la materia.

PickingRay

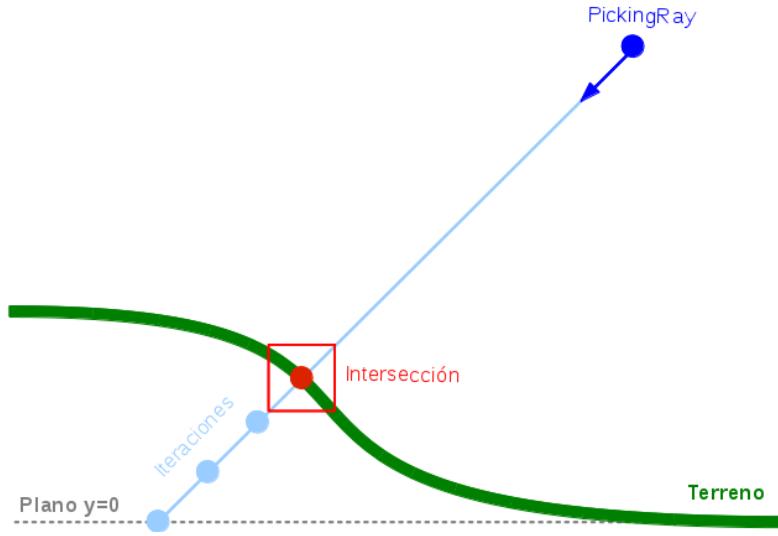
Intersección con HeightMap

Para hallar la intersección entre el Picking Ray y el HeightMap, simplemente buscamos el punto del rayo que coincide con un punto de la malla del terreno.

Este proceso es relativamente fácil gracias a una función del HeightMap que nos devuelve valores de Y dados un par X, Z. Hallar la intersección se reduce entonces a fuerza bruta: variamos los valores de X y Z en la ecuación del rayo y encontramos el valor para el cual (X,Y,Z) coincide para ambos elementos.



Sin embargo esto trae problemas de rendimiento que se notan a simple vista: buscar valores para (X,Y,Z) desde el origen del rayo recorriendo su vector dirección resulta poco performante. Esto se debe a que el origen se encuentra donde está la cámara, y el punto intersección tiende a estar ubicado en el nivel Y=0 del terreno. Es decir que estamos recorriendo la recta desde la cámara hasta el plano más bajo del HeightMap, y esto requiere una gran cantidad de iteraciones. Para solucionar esto lo que hacemos es recorrer la recta en el sentido contrario, es decir, desde el plano más bajo del HeightMap hacia "la pantalla". Y dado que, como ya mencionamos, el punto de intersección suele ubicarse más próximo a ese plano más bajo, la cantidad de ciclos que salvamos es abismal.



Movimiento por picking

Utilizando el principio anteriormente mencionado para hallar la intersección PickingRay-HeightMap, implementar movimiento por picking se reduce solamente a desarrollar la lógica del personaje para dirigirse a ese punto intersección.

Cilindro como volumen simplificado de personaje

Para representar mejor el volumen de cada personaje de manera simplificada evitando el uso de BoundingBox, se desarrolló la clase BoundingCylinder. Sus ventajas, desventajas, y detalles de implementación se explicarán más adelante.



Selección múltiple

Selección múltiple utilizando TgcBox

Utilizando el Picking Ray, y hallando la intersección de éste con el HeightMap, es fácil encontrar los puntos máximos y mínimos de la TgcBox que tiene inicio en el punto donde el usuario hizo click, y varía su tamaño para ajustarse a la posición actual del mouse. Luego, la selección de personajes se resuelve buscando colisiones entre sus cilindros y esta caja (como se explicará más adelante).

Selección múltiple utilizando un rectángulo proyectado en pantalla

Utilizando las coordenadas del mouse es simple dibujar un rectángulo cuyos puntos mínimo y máximo son la posición inicial del mouse antes de presionar el botón izquierdo, y la última posición, respectivamente. Para dibujar el rectángulo simplemente definimos los cuatro vértices que formarán los dos triángulos necesarios para pintar el área, aplicando transparencia.

Cuando el botón es soltado, la selección de personajes se realiza proyectando el cilindro de cada uno en pantalla (como se explicará más adelante), formando rectángulos. Filtrar los que están dentro del área de selección se reduce, finalmente, a encontrar una intersección entre estos y el rectángulo que dibujamos con el mouse.

Selección simple utilizando PickingRay

La selección simple se utiliza cuando el usuario hace un click en un solo punto de la pantalla, sin desplazar el mouse. Por ende, no se puede generar un rectángulo o proyectar una TgcBox. Es por eso que optamos por utilizar el PickingRay y buscar el personaje cuyo cilindro intersecta a este (proceso que se explicará más adelante).

BoundingCylinder

Dada la naturaleza del juego, donde los objetos y personajes se colocan y desplazan sobre el terreno (HeightMap) como si fuera un plano, y solo colisionan lateralmente, resultaba conveniente el desarrollo de un nuevo volumen simplificado que permita una mejora de rendimiento en el cálculo de colisiones.

Es por esto que decidimos que el cilindro era la mejor opción.

Composición

El cilindro tiene como atributos internos:

- Centro
- Radio
- Vector “HalfHeight”: vector (0, 1, 0) multiplicado por la mitad de la altura del cilindro.

Detección de colisiones

Dado que, como ya mencionamos, todos los objetos del juego permanecen sobre el terreno, el cálculo de colisiones entre un cilindro y otras figuras geométricas se puede reducir proyectando todo en el plano XZ. Entonces tanto un cilindro como una esfera se transforman en un círculo, y los BoundingBox en rectángulos. A partir de este punto todas las colisiones se reducen a encontrar intersección entre dichas proyecciones.

Colisión cilindro-cilindro y cilindro-esfera

Detectar una intersección cilindro-cilindro o esfera-cilindro se vuelve un caso trivial de intersección círculo-círculo, en donde se verifica que la distancia entre ambos sea menor a la suma de sus radios.

Colisión cilindro-BoundingBox

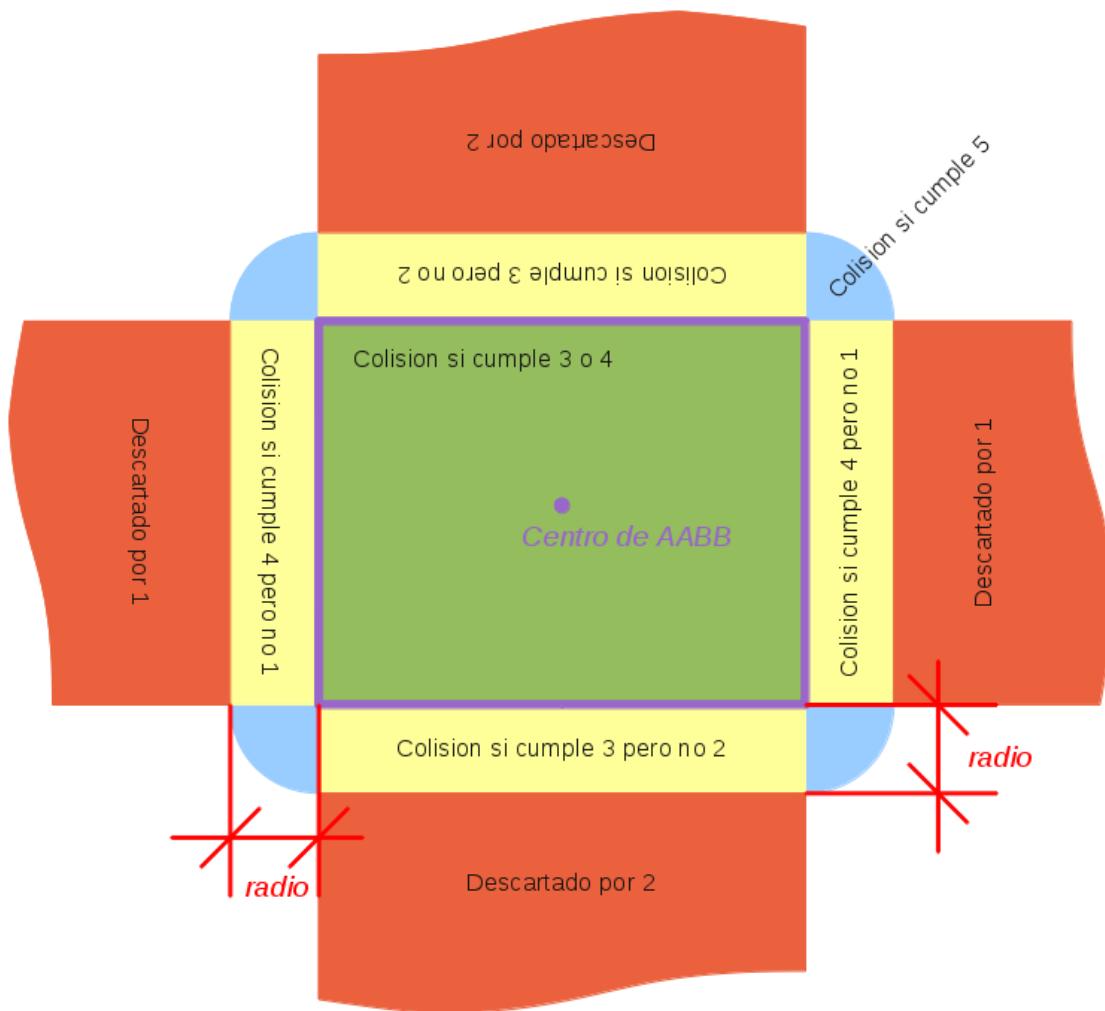
El caso más complicado se da en una colisión cilindro-BoundingBox. Las proyecciones resultan en un círculo para el cilindro y un rectángulo para el BoundingBox.

Para detectar colisión analizamos las posiciones de ambos en una serie de pasos:

1. Si la distancia en X entre ambos centros (del rectángulo y del cilindro) menos el radio es mayor a la mitad del tamaño en X del rectángulo, no hay colisión.
2. Análogamente, repetimos el paso 1 pero en el eje Z.
3. Si la distancia entre ambos centros en X es menor a la mitad del tamaño del rectángulo en X (es decir que su centro está ubicado dentro de los límites del rectángulo), hay colisión.
4. De igual manera repetimos el paso 3 pero sobre el eje Z.
5. Si no se dio ninguno de los casos anteriores, es posible que el cilindro colisione con una esquina del BoundingBox, en lo que se presentaría como el caso más complicado de

analizar. Lo que se hace es buscar la distancia entre la esquina y el centro del cilindro, y se la compara al radio de este. Si es menor, hay colisión. Todo ese proceso, en términos matemáticos, se hace restándole las dimensiones del rectángulo al vector distancia de centro a centro. Finalmente se obtiene el módulo de ese resultado para compararlo con el radio.

El siguiente gráfico muestra las posibles posiciones del centro del cilindro, y los resultados de la prueba de colisión dependiendo de cuales de las reglas anteriormente descritas cumpla dicha ubicación.



En violeta se muestra la proyección del AABB en el plano XZ. En verde las posiciones que implican colisión. En amarillo las que implican colisión pero que son detectadas tras varios pasos. En celeste las que llegan y dependen del paso 5. El resto, implica que no existe intersección entre ambos cuerpos.

Intersección con PickingRay

Para saber si el PickingRay atraviesa el cilindro, simplemente nos fijamos si el rayo atraviesa el plano longitudinal a éste, dentro de sus límites.

Hallar el plano es bastante simple: tomamos el vector dirección del rayo y lo post multiplicamos por el vector “HalfHeight” del cilindro. Este resultado lo pre multiplicamos nuevamente por este último vector, y así obtenemos la normal del plano. Para hallar la constante D utilizamos como punto el centro del cilindro.

Luego buscamos el punto del rayo que pertenece al plano. Finalmente saber si el PickingRay interseca con el cilindro se reduce a verificar que dicho punto está dentro de los límites de este.



Proyección en pantalla

Para la proyección en pantalla del cilindro realizamos el mismo procedimiento que los BoundingBox. Tomando el plano anteriormente descrito y necesario para hallar intersección con el PickingRay, usamos sus cuatro esquinas (dentro de los límites del cilindro) y las utilizamos como “puntos extremos” para proyectarlas en pantalla. Finalmente, el rectángulo se obtiene buscando valores mínimos y máximos entre las 4 proyecciones.



“Punto del cilindro más cercano”

Para implementar otras funcionalidades del juego se necesitaba hallar el punto perteneciente al cilindro más cercano a determinado punto.

Se trata de un proceso muy simple: si el punto especificado está dentro del rango del cilindro en el eje Y (es decir, entre las tapas del cilindro), el valor buscado se encontrará en la superficie lateral del cilindro, en la dirección de dicho punto. Solo hay que hacer la salvedad de que, si ese valor está fuera del rango en Y, el punto estará contenido en una tapa.

Cono de visión

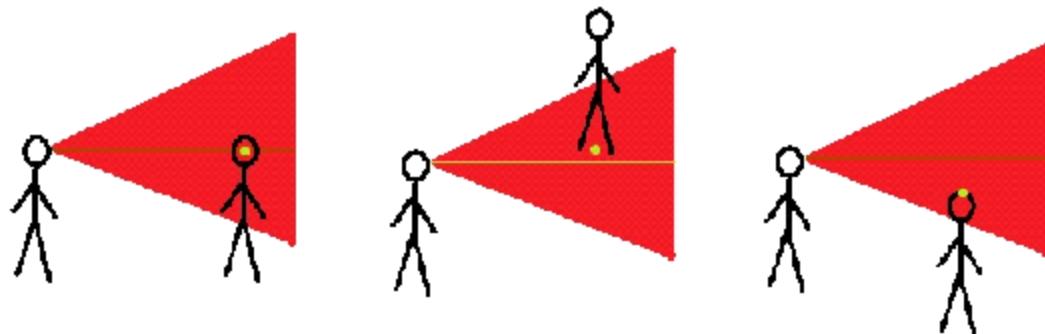
Los soldados enemigos detectan a los Comandos cuando éstos están dentro de su área de visión. Un modelo bastante realista para modelar el área de visión de los personajes es mediante un cono que parte desde el personaje en la dirección en la que está mirando.

Geometría

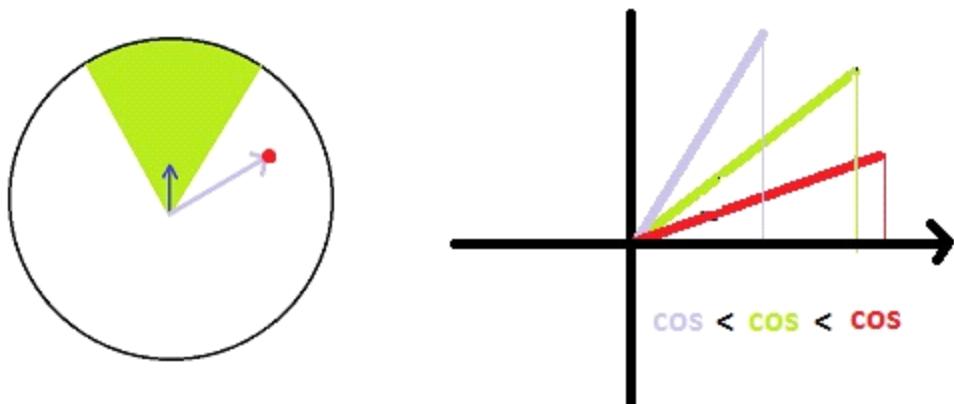
El cono se define por un vértice V, un largo L, un ángulo A y una cantidad de triángulos T. A partir de L y A se calcula el radio de la circunferencia que se encuentra a distancia L del cono. Se crea una circunferencia con T puntos y luego se crean los triángulos (formados por el vértice y dos puntos de la circunferencia). Ver clase Cone.cs

Funcionamiento

Para saber si un jugador está dentro del cono, se calcula si el punto del eje del jugador que está más cerca del eje del cono está dentro del cono. Inicialmente este cálculo se hacía con los extremos del bounding box, pero el cálculo era más costoso y se daban casos en los que el bounding box tocaba el cono y el comando no. Las pruebas mostraron que para lo que requería el juego, bastaba con un sólo punto.



Para saber si un punto cae dentro del cono, primero se verifica que el punto se encuentre a distancia L_0 menos del vértice del cono (se comparan los cuadrados de las distancias para no tener que realizar raíces cuadradas), luego se realiza el producto escalar del versor dirección del cono y el versor que va del vértice del cono hacia el punto, obteniéndose el coseno del ángulo entre ambos versores.



Cuanto mayor sea el ángulo, menor será el coseno. Por lo que el punto estará dentro del cono si y sólo si el coseno es mayor al coseno del ángulo de apertura del cono.

Oclusión

Para saber si hay un obstáculo, se lanza un ray desde el vértice del cono hasta el punto elegido, testeando colisión sólo con los objetos del nivel que se encuentran en un radio menor o igual al que va desde el vértice hasta el punto del comando.

También se verifica que la altura del terreno no exceda la de la vista en ningún punto entre el comando y el enemigo.

Rango de visión

Cuando el comando se encuentra agachado, los cálculos se realizan con la mitad del radio del cono.

IA general

Tanto los Comandos como los soldados enemigos utilizan el método Character::goToTarget(Point). Los Comandos lo usan para ir hacia el punto de picking y los enemigos para dirigirse a un waitpoint o perseguir Comandos.

Se obtiene la dirección del movimiento y se la multiplica por la velocidad para obtener la nueva posición. En caso de que el terreno sea muy empinado se cancela el movimiento.

Tras eso se realiza el movimiento y se testea colisión esférica con los objetos del nivel, luego se testea colisión cilíndrica con los objetos que anteriormente colisionaron. En caso de chocar con un objeto el personaje retorna a la posición previa al movimiento y se verifica si se trata del objetivo (en caso de que se haya seleccionado un personaje) o de si se chocó contra alguien que está sobre el objetivo. De ser así se considera que el personaje llegó a su destino. En otro caso se intenta esquivar el objeto. El algoritmo es sencillo: se obtiene el vector que va del centro del objeto al centro del personaje, se suma a la dirección del movimiento, se normaliza y se intenta realizar el movimiento nuevamente con esta nueva dirección. El procedimiento se repite hasta que no haya colisión o se haya superado el máximo de intentos, si se da el segundo caso se cancela el movimiento. El resultado es un movimiento circular alrededor del obstáculo.

IA de personajes enemigos

Los personajes enemigos (como mencionamos anteriormente) poseen un cono de visión e implementan la IA necesaria para detectar si un Commando está frente a él.

Pero más allá de esto, también implementan la IA de patrullaje y la IA de persecución, que se detallan a continuación.

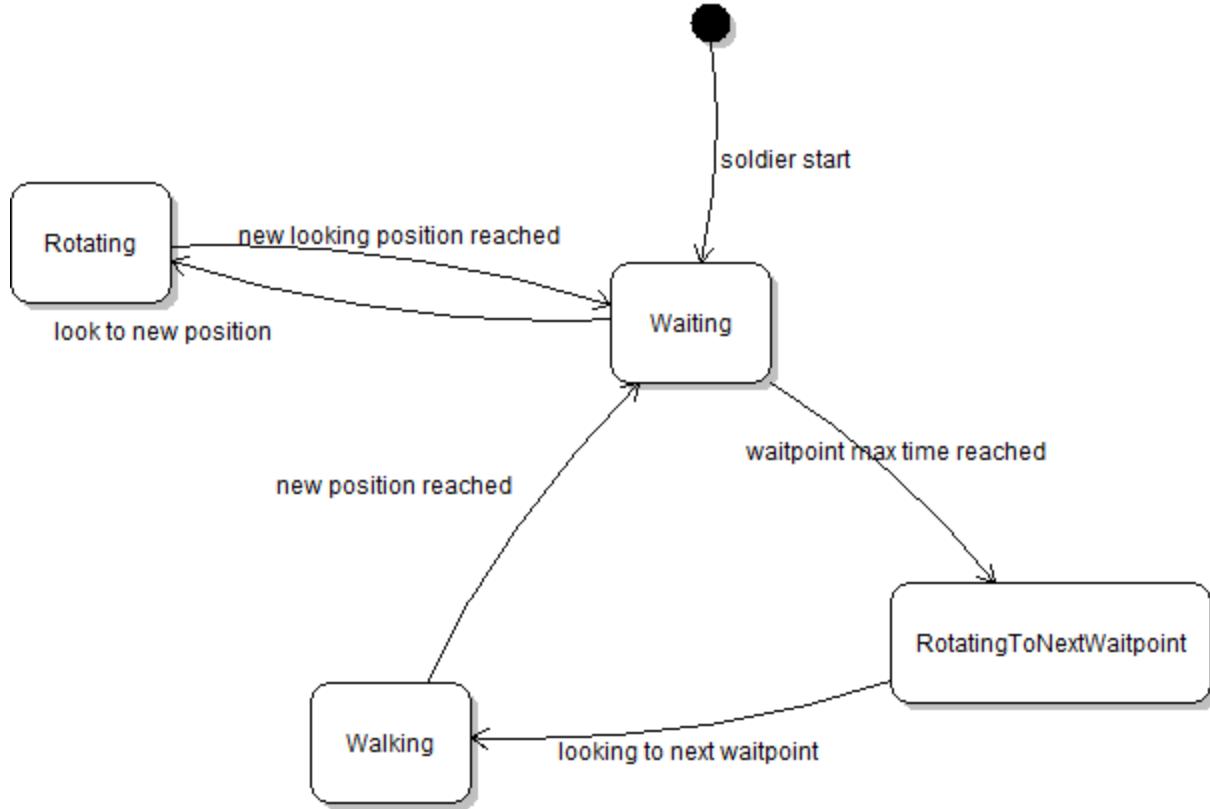
IA de patrullaje con waitpoints

Implementado como un State, la IA de patrullaje intenta generar movimientos fluidos de un enemigo al recorrer sus distintos waitpoints.

Cuando el enemigo está en un waitpoint, va rotando en busca de Comandos, alternando entre los States Waiting y Rotating.

Luego de determinado tiempo esperando, gira hacia la posición del siguiente waitpoint, antes de empezar a desplazarse hacia él. Eso es responsabilidad del State RotatingToNextWaitpoint, que luego de dejar el personaje alineado con la dirección pasa a Walking.

Cuando llega a un nuevo waitpoint, el ciclo se repite indefinidamente.



El único suceso que corta este ciclo de ejecución es la aparición de un Commando en el rango de visión del enemigo.

IA de persecución

En el momento en que un enemigo ve un Commando comienza la persecución (State Chasing), en cada update la posición del Commando es seteada como objetivo y se utiliza el método `goToTarget`. A su vez, se descuenta vida del Commando. Tanto si el Commando muere como si se sale del área de visión, el enemigo se dirige a el último punto en el que fue visto y pasa a estado Waiting con alerta (se volteá más rápido y más seguido).

Imágenes 2D

Para las imágenes 2D (Mapa, vidas, panel) se creó una clase Picture. Se utilizó dicha clase en lugar de Sprite para tener un mayor control sobre la implementación. Esta clase renderiza la imagen sobre un rectángulo. El primer canal se utiliza para renderizar el diffuse map, mientras que el segundo se utiliza para opacity map y marcos. Esto permite que las coordenadas del diffuse map difieran de las del opacity map y el marco, lo cual es útil a la hora de hacer zoom en el mapa. El marco se renderiza luego de haber renderizado la imagen.

Opacity map

El shader es similar al utilizado en un lightmap, la diferencia es que en lugar de modificar el color lo que se modifica es el alfa.



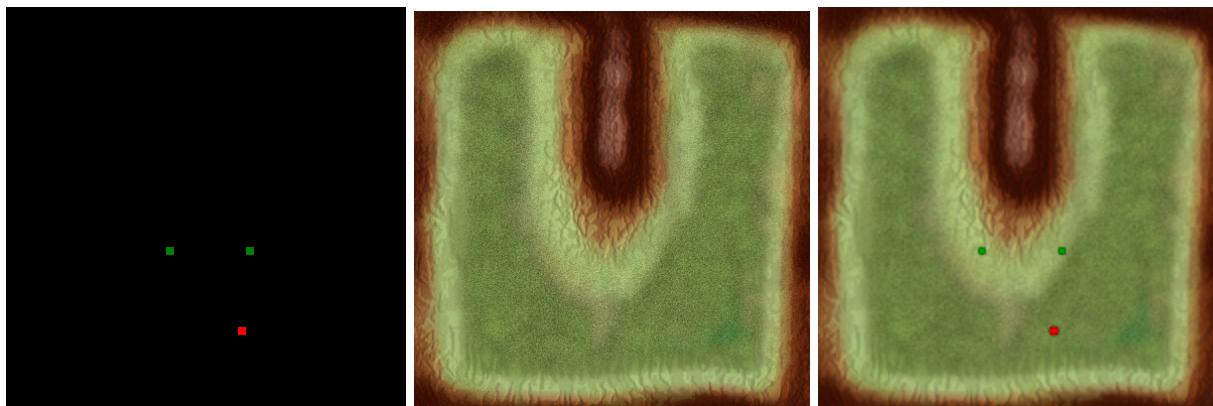
Mapa

El mapa hereda de Picture, conoce al nivel y por lo tanto al terreno y todos los personajes. El primer canal de textura se utiliza para la imagen del mapa propiamente dicha, sus coordenadas se modifican según la posición de la cámara y el zoom elegido.

La secuencia de renderizado es la siguiente:

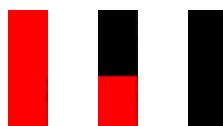
1. Se crea un rectángulo por cada jugador, si es un enemigo el color de los vértices es rojo, caso contrario verde.
2. Se cambia el render target.
3. Se renderizan los rectángulos, obteniéndose una textura negra con rectángulos de colores.
4. Se restaura el render target.
5. Se realiza el render pasando al shader como parámetros el diffuse map, opacity map y las posiciones de los jugadores.

6. Por cada fragmento, si el color del pixel de la textura de posiciones es distinto de negro, se retorna ese color, caso contrario se usa el color del diffuse map (textura del terreno). Antes de retornar el pixel, se setea la opacidad que indique el opacity map.



Barras de vida

Para la vida de los personajes se utilizó un pixel shader al que se le pasa como parámetro la relación entre la cantidad de puntos actual y la total, tanto en x como en y (Si la barra es vertical, **Y** vale 1-puntos/total y **X** vale 1, mientras que si es horizontal **X** vale puntos/total e **Y** vale 0). Se compara la coordenada de textura con la proporción de puntos y si la coordenada en **U** es menor o igual a la proporción de puntos en **X** y la coordenada **V** es mayor a igual a la proporción de puntos en **Y**, se propaga el color. Caso contrario el pixel será negro.



DivisibleTerrain

Se creó una clase basada en `TgcSimpleTerrain` que está compuesta por un conjunto de `TerrainPatch`. En lugar de utilizar un sólo vertex buffer, se modificó el ciclo de creación de los triángulos para que se carguen en varios vertex buffer, y con cada uno se crea un `TerrainPatch`. Los `TerrainPatch` pueden deshabilitarse, lo cuál permite aplicar técnicas de culling sobre el terreno. También se agregaron métodos que retornan la altura del terreno.

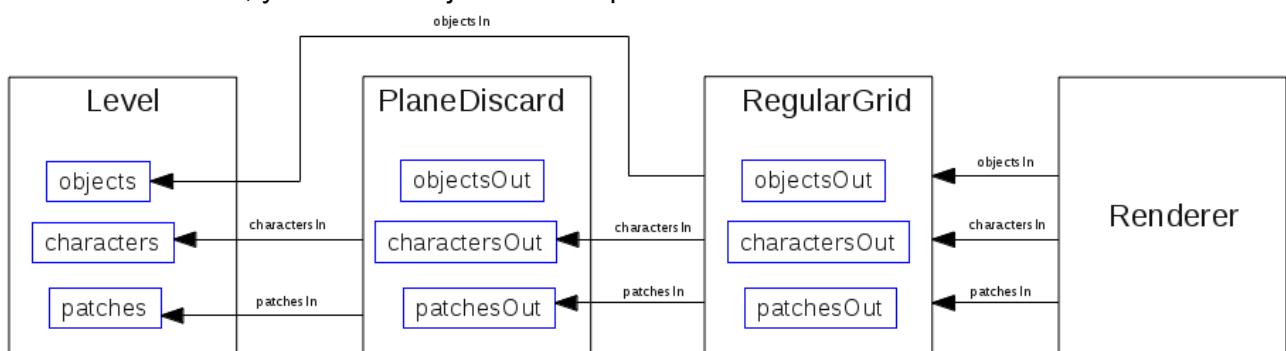
Optimización

Técnicas encadenadas

Para poder aplicar múltiples técnicas de optimización, implementamos una “cadena de filtros”. Inicialmente es el Nivel quien conoce a todos los personajes, objetos, y sectores del terreno que hay que renderizar. Existe un objeto `Renderer` que es el que se encarga de ello. Este objeto conoce las tres colecciones (personajes, objetos y sectores del terreno), y en cada ciclo itera sobre sus componentes renderizándolos uno a uno.

Ahora bien, lo que hicimos fue desarrollar una serie de objetos que trabajan de manera muy similar al `Renderer`: conocen 3 colecciones de entrada, y tienen 3 colecciones de salida. Cada uno tiene un algoritmo propio, que se encarga de llenar sus colecciones o listas de salida. Es así como, seteando a cada uno para que le ingrese lo que sale del anterior, podemos encadenar técnicas de optimización.

En nuestro caso, implementamos dos técnicas de optimización. La primera es el “`PlaneDiscard`”, que aplica sobre los personajes y los sectores del terreno que posee el nivel. Luego filtramos utilizando la Grilla Regular, aplicándola sobre los personajes y sectores salida del “`PlaneDiscard`”, y sobre los objetos del mapa.



Este orden de aplicación está dado por la facilidad de la técnica `PlaneDiscard` para eliminar muchos sectores con poco esfuerzo antes de llegar a la Grilla Regular. Dependiendo de la posición de la cámara, se pueden descartar de entrada más de la mitad de los sectores y personajes del mapa, lo que aliviana mucho los testeos de colisión frustum-aabb que va a

realizar RegularGrid.

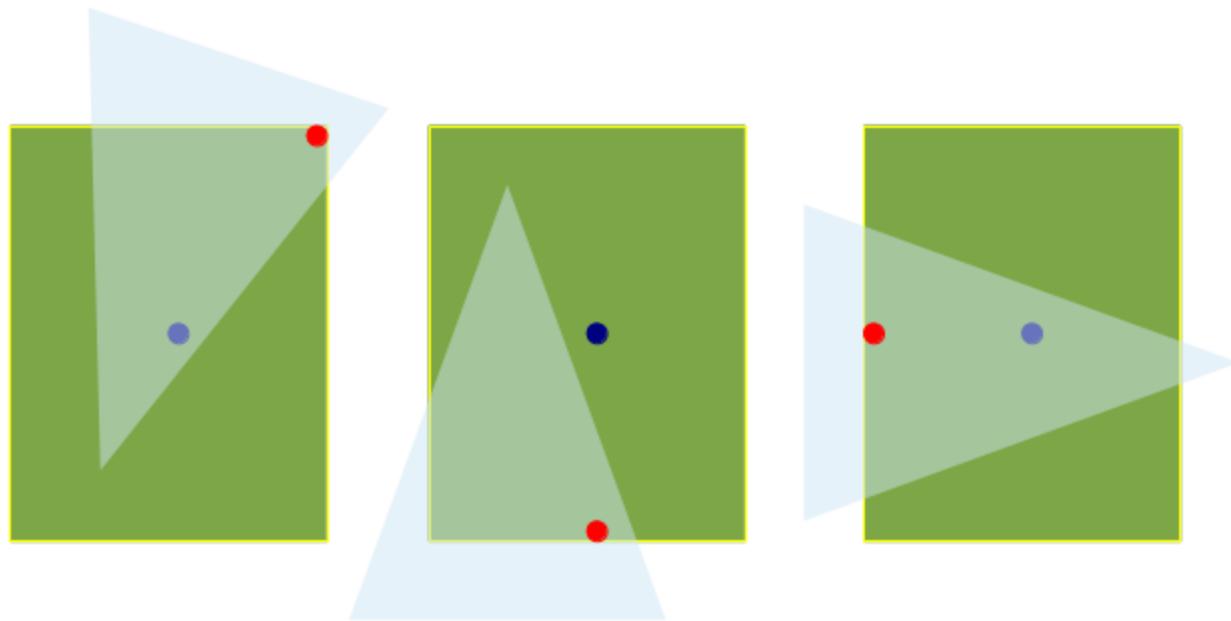
PlaneDiscard

Esta técnica filtra todos los elementos ubicados por detrás del plano de la cámara. La normal de dicho plano es la dirección en la que se está mirando.

Su algoritmo es muy eficiente, ya que solo toma las posiciones de los elementos y se fija si ese punto está por detrás del plano.

Dado que la posición de un personaje “representa fielmente” su ubicación (y volumen), esta técnica se presenta como ideal para filtrarlos. Pero para objetos y sectores del terreno existen ciertas complicaciones: dado que los objetos más grandes no pueden ser representados con un solo punto, es normal que “desaparezcan” cuando la cámara sobrepasa su centro pero no su volumen total. Dado que solucionar esto requiere una serie de cálculos “extra” en cada ciclo (desvirtuando la idea de un algoritmo liviano), directamente no utilizamos esta técnica para filtrar objetos.

El mismo problema se presenta para los sectores del terreno, pero en este caso, la solución es más simple: en lugar de tomar el centro del sector, utilizamos el punto “más adelantado respecto de la cámara”, como se observa en los siguientes gráficos.



En azul, centro del sector. En rojo, punto que se toma como referencia.

Grilla Regular

Esta técnica utiliza el frustum de la cámara para hallar colisiones frustum-aabb con el BoundingBox de cada sector del terreno.

Como también filtra objetos, se guarda de manera offline una lista de sectores con los objetos que contiene cada uno, buscando colisiones entre ambos. Es claro que un objeto puede pertenecer a varios sectores, y esto podría generar que se descarte por error. Pero hay que recordar que cada filtro no descarta elementos sino que los agrega a sus listas de salida. Es por este detalle que los filtros fueron diseñados así.

Volviendo a la técnica en sí, en cada cuadro busca colisiones entre el frustum y los sectores del terreno de la colección de entrada. Cuando existe colisión, agrega el sector y sus objetos a sus listas de salida.

En cuanto a los personajes, dado que (como mencionamos anteriormente) su posición “representa fielmente” su volumen, simplemente itera uno a uno y se fija si se encuentran dentro de un sector a renderizar. Esto lo hace escatimando cálculos: en lugar de testear colisión, proyecta tanto el punto como el AABB en el plano XZ, y corrobora que dicho punto se encuentre dentro del área del rectángulo.

Sombras

Para las sombras se modificó el shader ShadowMap.fx agregándole funciones que realizan skinning para así poder utilizarlo también con TgcSkeletalMesh. Para ir alternando entre modos de renderizado (Con Sombras, Sin Sombras) hicimos que Level tenga un strategy Renderer, que conozca los objetos a renderizar, la información necesaria para el efecto (ej: posición de la luz), settee las técnicas que correspondan y realice los renderizados necesarios (ej: render del shadow map), en el orden que corresponda.

Para variar la posición de la iluminación se implementó la noción de día truncando el elapsedTime del juego a 3 minutos. El 20% del tiempo es noche absoluta, con el sol ubicado en un punto fijo externo al SkyBox. El tiempo restante, el sol se mueve linealmente respecto al heightmap (coordenadas x-z), y describe una parábola con concavidad negativa en el eje Y.

Dado que la iluminación siempre apunta al centro del terreno, el resultado final es una parábola que recorre el escenario de una punta a la otra, simulando el movimiento perceptible del sol desde la Tierra.

Archivo de configuración

Los niveles se crean mediante un archivo XML.

default-level.xml

<level>

```
<terrain
  heightmap="Heightmaps\\heightmap.jpg"
  texture="Heightmaps\\TerrainTexture5.jpg"
  scaleXZ="20.0"
  scaleY="2.0"
  format="(3,3)" Opcional. Son las divisiones del terreno. El valor default es (1,1)
></terrain>
```

El mínimo de enemigos es 0.

<enemy>

El mínimo de waitpoints es 1.

```
<waitpoint>(460,620)</waitpoint>
<waitpoint>(-560,600)</waitpoint>
<waitpoint>(-800,-260)</waitpoint>
</enemy>
```

La picture es opcional.

```
<commando picture="\\CharacterPictures\\1.jpg">(-200,200)</commando>
<commando picture="\\CharacterPictures\\avatar_2976.jpg">(200,200)</commando>
```

```
<levelObject class="tree" scale="(5,7,5)">(56,530)</levelObject>
<levelObject class="wall" size="(500,60,50)">(200,700)</levelObject>
```

También se pueden cargar meshes:

```
<levelObject mesh="ArbolSelvatico2\\ArbolSelvatico2-TgcScene.xml"
scale="(5,5,5)"></levelObject>
```

```
</level>
```

Para la creación del level se utiliza la clase XMLLevelBuilder , por cada tag hay una clase XMLtag que se encarga de leer los atributos del nodo y devolver la clase correspondiente. La idea era tener varios tipos de comando y enemy pero no alcanzó el tiempo de desarrollo, por lo que si bien se puede utilizar el atributo class, las únicas clases posibles son *commando* (para commandos) y *soldier* (para enemy). No es necesario especificarlo en el xml porque son el valor default. En caso de agregar más clases sólo habría que modificar las clases XmlCommando y XmlEnemy para que retornen una clase diferente.