TGC 2013
K3051
Miercoles Noche

Trabajo Grupal
Fisica del auto
Los Borbotones

Los Borbotones

Nombre y Apellido	Legajo
Carolina Galassi Borba	144.334-3
Jonathan Corallo	144.522-4
Juan Jacobs	144.363-0
Javier Sorella	120.236-4

Fecha de Presentación	
Fecha de Devolución	
Calificación	
Firma Profesor	

Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

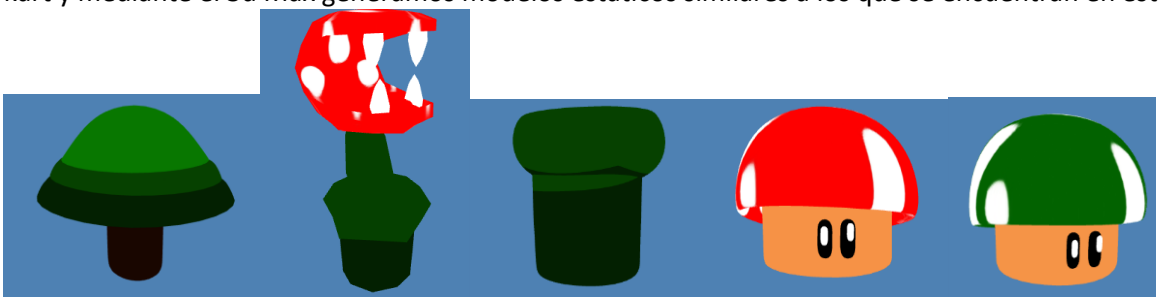
Índice

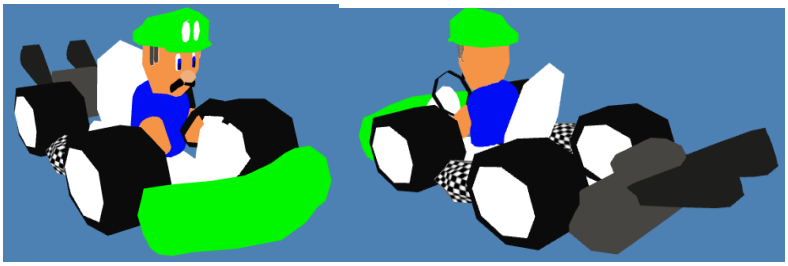
El **objetivo** del juego es recolectar todos los checkpoints dispuestos en el nivel antes de que el tiempo llegue a cero. De hacerlo se dispondrá la pantalla de juego ganado y en caso contrario de juego perdido. Tambien se dispondrá la pantalla de juego perdido en el caso que se acabe el tiempo o la vida del personaje debido a las colisiones de este

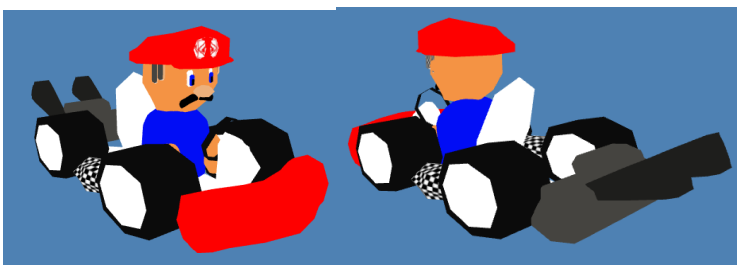
Teclado y controles

- **W** acelera
- **S** desacelera
- **A** dobla a la izquiera
- **D** dobla a la derecha
- **M** música y sonidos on/off
- **Q** vuelve al menu principal
- **R** vuelve a la posición inicial
- **J** carga al jugador seleccionado
- **B** Debug mode (carga OBBs y otros datos útiles)
- **I** Modo Dios

Decidimos darle una ambientación al ejemplo creativo inspirándonos en el tan conocido Mario kart y mediante el 3d Max generamos modelos estáticos similares a los que se encuentran en este.







El trabajo consta de cuatro grupos de objetos principales:

- Autos
- Niveles
- Pantallas
- Imágenes y Sonidos

Los **autos** tienen como atributo más importantes su velocidad máxima, velocidad de rotación, velocidad actual, masa, aceleración y dirección. Estos datos son utilizados a la hora de simular la física de movimiento y las colisiones contra objetos. Por otro lado, también tiene un mesh diseñado por el grupo. En esta clase se encuentra la lógica referida al movimiento y deformación de los mismos, ambas explicadas más adelante en el documento. Un jugador puede elegir entre dos autos distintos a la hora de empezar a jugar.

Los **niveles** tienen la información que respecta al escenario de juego. Al estar simulando la física de autos, optamos por modelar una pista de carreras. Para hacer el piso se utilizó un TgcBox sin altura –es decir, plano- con una textura, que es la imagen de la pista. Para simular el espacio de juego se utilizó un TgcSkyBox, con texturas que son imágenes del cielo.
Los niveles tienen como atributo una colección de obstáculos y una de checkpoints. En esta clase, se posicionan los distintos obstáculos y checkpoints a lo largo del escenario y se controla el renderizado de cada uno de ellos, según la técnica de optimización elegida, explicada en este documento.

Las **pantallas** que hay en el juego son tres:

Pantalla de inicio: El juego inicia con esta pantalla, en la que se muestran los personajes a elegir para jugar. Consta de sprites 2D.

Pantalla de juego: Es la pantalla de juego, desde que se eligió un personaje hasta la finalización del juego. En ese momento se renderiza el escenario, el auto del personaje elegido, los obstáculos y los checkpoints. Además, se muestra una barra de vida del personaje, puntos acumulados y un cronómetro de cuenta regresiva que indica el tiempo restante para finalizar el juego.

Pantalla de finalización: Cuando se llega al fin del juego, se utiliza esta pantalla. En caso de que el jugador se haya quedado sin vida o sin tiempo antes de llegar a pasar por todos los checkpoints, se muestra una pantalla de game over. En caso contrario, se muestra una pantalla de victoria. En ambos casos se ofrece la posibilidad de volver a jugar.

Colisiones

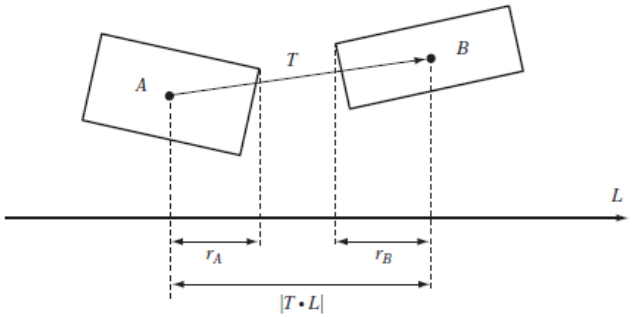
Los objetos colisionables son todos aquellos con los que un auto puede, como su nombre lo indica, colisionar. Más particularmente, en nuestro trabajo práctico, son colisionables los checkpoints y los obstáculos. Como sería muy costoso hacer test de colisión para los modelos triángulo a triángulo, utilizamos el volumen de simplificación OBB (Oriented Bounding Box). Este volumen fue elegido por sobre los AABB (Axis Aligned Bounding Box) y las Bounding Spheres debido a que por la naturaleza de los movimientos del auto, se necesitó un volumen de simplificación que pueda ser rotado junto con el auto. En el juego, el auto nunca cambia de altura, sino que está siempre apoyado en el plano del piso. Esto también simplifica los test de colisiones y los cálculos de respuestas, ya que aunque se está en un universo de tres dimensiones, las colisiones se pueden considerar en dos dimensiones.

Para modelar los objetos colisionables, utilizamos la clase ObstaculoRigido y la clase Checkpoint, que constan de un mesh o TgcBox y su respectivo OBB. La diferencia entre un checkpoint y un obstáculo –más allá del dominio juego- es el tratamiento de las colisiones que tiene cada uno.

Cada vez que un auto colisiona con un checkpoint, este último deja de dibujarse en pantalla y habilita el renderizado del siguiente.
En cambio, cuando un auto colisiona con alguno de los distintos obstáculos, se produce un rebote acorde a la velocidad que se tenía al colisionar, una desviación/rotación acorde al ángulo de incidencia del auto en el obstáculo, una disminución de velocidad y, si al momento de colisionar el auto estaba yendo a una velocidad lo suficientemente grande, se produce una deformación en la carrocería con expulsión de chispas, junto con la reproducción de un sonido cuyo volumen varía respecto a la velocidad de choque. Además, cada impacto reduce la velocidad máxima del vehículo, proporcionalmente a la velocidad del impacto.
La velocidad del auto se conoce en todo momento.

Intersección entre OBBs

La prueba de solapamiento entre dos OBBs es sorprendentemente complicado. En su forma más simple, esta prueba se podría realizar mediante la comprobación de si los vértices de un cuadro están todas en el exterior de los planos definidos por las caras de la caja B, y viceversa. Sin embargo, este trabajo de prueba en 2D no funciona correctamente en 3D. No es capaz de hacer frente a, por ejemplo, el caso en el que A y B están casi en contacto borde con borde, los bordes perpendiculares entre sí. Aquí, ninguna caja está completamente fuera de toda una cara de la otra. Por consiguiente, la simple prueba de ellos informa que intersectan a pesar de que esto no ocurre.



Una prueba exacta de OBB-OBB intersección se puede implementar en términos de lo que es conocida como la prueba del eje de separación.: dos OBBs están separados si, con respecto a unos ejes L, la suma de sus radios proyectada es menor que la distancia entre la proyección de sus puntos centrales. Es decir, si

$$| T \cdot L | > | rA + rB |$$

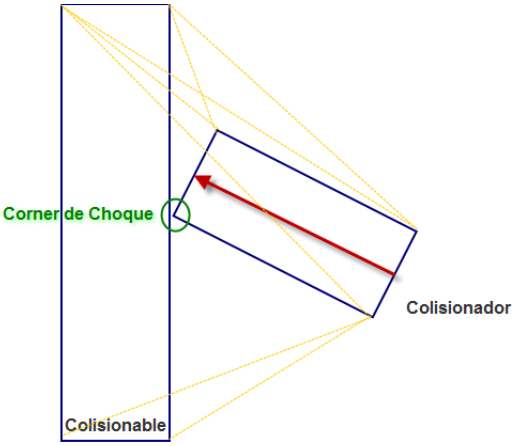
Para OBBs es posible demostrar que en la mayoría de estos 15 ejes que separan, se probarán

para determinar correctamente el estado de superposición OBB. Estos ejes corresponden a los tres ejes de coordenadas de A, los tres ejes de coordenadas de B, y los nueve ejes perpendiculares a un eje de cada uno. Si las cajas fallan en superponerse en cualquiera de los 15 ejes, no se están intersectando. Si no hay ningún eje que cumpla este “early out”, entonces las cajas se están solapando.

Nuestra implementación es una mejora del modelo presentado en el libro Real Time Collision Detection de Christer-Ericson.
Este algoritmo se puede analizar en el método testObbObb2() dentro de la clase Colisiones.cs.

Deformación de la Carrocería

El principal inconveniente que tuvimos con la deformación fue calcular el punto de choque preciso para que a partir de allí el punto a deformar en el mesh sea el correcto.

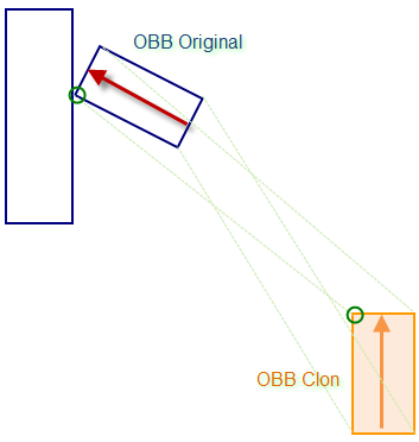


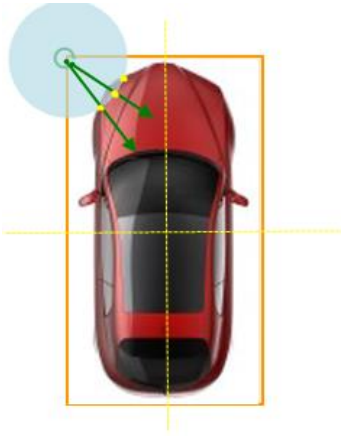
Cuando se produce una colisión del coche contra un obstáculo, primero computamos todos los corners del OBB del vehículo y los comparamos con todos los corners del obstáculo para medir la distancia entre ellos. La distancia mínima resultante determinará cual será el corner del vehículo más cercano al objeto colisionado.

Ahora necesitamos aplicar una deformación dentro del mesh a partir de este corner. Para eso, tomamos el verterBuffer del Mesh, esto es, el conjunto de vértices del Mesh tal como está definido en el XML.
Aquí estamos frente a un primer inconveniente, ya que la posición de los vértices están en el lugar donde se cargó al Mesh por primera vez. Como el vehículo se traslada y rota según como se traslade y rote el OBB, los vértices del vertexBuffer estarán distantes al punto de choque.
Para esto existen dos soluciones posibles:

1) Trasladar y rotar los vértices al punto de choque
2) Trasladar el punto de choque donde se encuentran los vértices

Como trasladar vértice por vértice al punto de choque puede ser computacionalmente costoso, resulta mucho más sencillo trasladar el OBB donde se encuentra el vertexBuffer. Como no podemos trasladar el OBB sin afectar la continuidad del juego, lo que decidimos fue clonar el OBB y rotar y trasladar este clon al sitio del vertexBuffer.





Finalmente, podemos calcular a partir de este punto una deformación en el mesh. Una variable "factorChoque" determinará un radio de impacto. Este valor es proporcional a la masa del vehículo y a la velocidad del impacto. Mientras mayor el impacto, mayor será el radio de impacto, tomando así más vértices para desplazar dentro del mesh. Todos los vértices se desplazarán por el vector que se produce entre el punto de choque y el vértice candidato dentro del radio.

Este algoritmo se puede analizar en el método deformarMesh() de la clase Auto.cs.

Chispas

Cada vez producido un choque de gran magnitud, el auto mostrará chispas en el punto de choque al momento de chocar. El punto de choque será aquel vértice más cercano al corner de choque del Obb. En este sitio se sitúa una cantidad preestablecida de partículas (esferas) que se desplazarán a través de un vector cada vez producido el impacto. Mientras mayor sea el impacto, más tiempo se reproducirán las chispas.

Respuesta de Choque

Para calcular el ángulo de incidencia del auto en el obstáculo se realizaron los siguientes pasos:

- Se calculan los extremos de los OBB del obstáculo con el método "computeCorners" que se encuentra en la clase CalculosVectores
- En base a los extremos obtenidos, se calcula el plano correspondiente a cada una de las cuatro caras del objeto en las que es posible la colisión, utilizando el método "FromPoints" de la clase Plane de DirectX.
- Una vez obtenidas las caras, se busca entre ellas cuál es la más cercana al punto del impacto utilizando el método"distancePointToPlane" que se encuentra en la clase CalculosVectores.
- Cuando ya se sabe cuál es la cara en la que el auto chocó, se procede a hacer el cálculo del ángulo de incidencia. Esto se logra utilizando el vector dirección el auto al momento de chocar y el vector normal al plano que incluye a la cara chocada.

Física de movimiento

A fines de simplificar la física de movimiento del auto, se modeló como un movimiento uniformemente variado.

$$x(t) = \frac{1}{2}at^2 + v_0t + x_0$$

X(t): posición en función del tiempo.

a: aceleración

t: tiempo

Vo : velocidad inicial

Xo: posición inicial

En la clase auto se encuentran los distintos métodos que alteran la dirección y velocidad del mismo. Se cubre

- Aceleración
- Desaceleración
- Marcha atrás
- Frenado por inercia
- Doblar con derrape

Con el teclado se puede manejar el auto para lograr los diferentes movimientos. El fenómeno de aceleración se consigue reemplazando en la ecuación de MRUV el valor de la velocidad actual, el valor de la aceleración característica de cada auto y un valor Δt, que es equivalente al tiempo transcurrido desde que se apretó la tecla W.

La desaceleración y la marcha atrás se consiguen análogamente pero utilizando valores negativos. Se tuvo en cuenta que la desaceleración tiene que ser más rápida que la aceleración. Si acelero un auto durante 30 segundos, se tarda mucho menos que 30 segundos en desacelerarlo. En nuestro ejemplo, planteamos que el auto se desacelera 5 veces más rápido de lo que se acelera.

Para lograr el frenado por inercia, se buscan los momentos en los cuales el auto está andando pero el jugador no aplica aceleración ni desaceleración sobre el mismo. Cuando eso sucede, se le aplica al auto una desaceleración constante, menor a la de los frenos del auto, pero que eventualmente lo detiene completamente.

Para lograr el efecto de derrape al doblar, antes de renderizar el auto, se rota la malla un ángulo proporcional al tiempo transcurrido desde que se empezó a doblar, siendo menor a un ángulo de tope máximo a derrapar, después se renderiza la malla y luego se devuelve la malla al ángulo de rotación que tenía anteriormente para no alterar los próximos cálculos.

Al derrapar, se toma la velocidad de rotación con la que fue inicializado el auto y aumenta según la velocidad actual en la que se esté desplazando y el tiempo transcurrido que lleva doblando.

Optimización

Debido a la cantidad de modelos presentes en el escenario decidimos que era necesario optimizar la detección de aquellos modelos que fueran alcanzados por el frustum.

Decidimos aplicar el algoritmo Quadtree de los ejemplos del framework de la cátedra, ya que el modelo del auto solo puede desplazarse en X y Z, es adecuado particionar el escenario en celdas solo por estas dos dimensiones.

El algoritmo consiste en subdividir el escenario en celdas en forma jerárquica e iterativa, se sigue subdividiendo aquellas celdas que tengan una cierta cantidad de modelos en su interior, luego se detecta cuáles de estas celdas colisionan con el frustum para solo analizar la detección de los modelos contenidos por éstas en vez de todos los modelos del escenario, agilizando el cálculo.

Para el umbral de corte del algoritmo se tuvo en cuenta una cantidad de 5 modelos como máximo por celda y hasta una profundidad de nivel 2 como máximo para los nodos del árbol.

Ajuste de cámara

Para la configuración de la cámara se tuvo en cuenta que al realizar la acción de doblar, el ajuste se hiciera medio segundo después para que tuviera un retardo, y distintas velocidades de ajuste según la diferencia de rotación entre el modelo y la cámara.

Para lograr esta lógica se utilizó una variable tiempoTranscurrido, la cual aumenta según el tiempo que lleva doblando, con un tope de 1.5 segundos, y disminuye cuando se deja de doblar o se dobla en sentido contrario al que se venía, luego cuando no hay desfasaje entre el modelo y la cámara, la variable vuelve a 0.

Cuando un objeto se interpone en el segmento del auto y la cámara, se realiza un ajuste de cercanía de la visión, calculando el punto de intersección de dicho segmento con el bounding box del objeto que se interpone, se obtiene la colisión de menor distancia y luego se reposiciona la visión de la cámara desde dicho punto.

Motion Blur

Se emplea un shader de Gaussian Blur en toda la pantalla, el cual se activa a partir de cierta velocidad y aumenta gradualmente al llegar a la velocidad máxima del vehículo.

Modo Debug / Modo Dios

Para el testeo del proyecto, implementamos un Modo Debug. Activando este modo con la tecla “B”, podemos visualizar todos los OBB de los meshes en pantalla, el punto de choque (representado por una esfera gris), y una flecha ajustable por Modifiers, la cual nos sirvió para obtener las coordenadas de las posiciones donde deseábamos colocar los objetos. Esto nos ahorró mucho tiempo de trabajo y nos facilitó la detección temprana de errores.

El Modo Dios se activa con la tecla “I”. También para uso de debug, nos sirve para inhabilitar ciertas funciones de choque (la de perder energía) y congela el paso del tiempo.

Referencias

Real Time Collision Detection, Chapter 4 (Christer-Ericson, Morgan Kaufmann Publishers © 2005)