Entorno Prácticas

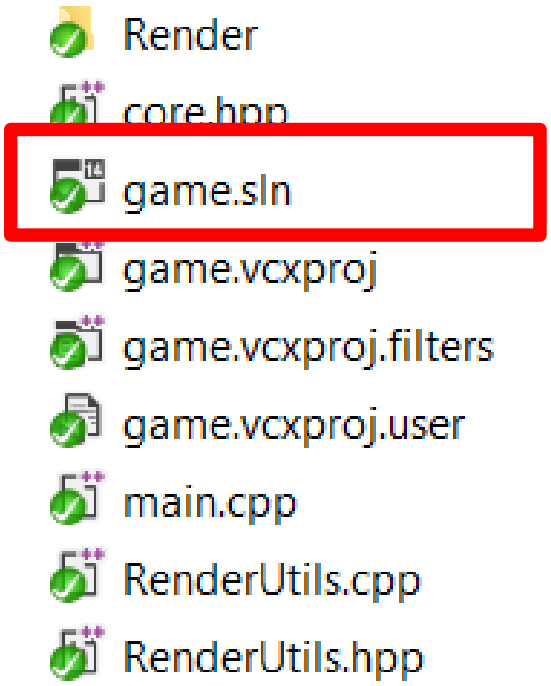
- Vamos a realizar las prácticas en C++
- Usaremos la librería PhysX de Nvidia

Lenguaje

- Vamos a realizar las prácticas en C++
- Usaremos la librería PhysX de Nvidia
- Empezaremos haciendo toda la simulación nosotros mismos
- Usando alguna clase de PhysX
  - Así no cambiamos de entorno cuando empecemos a usarlo para simular

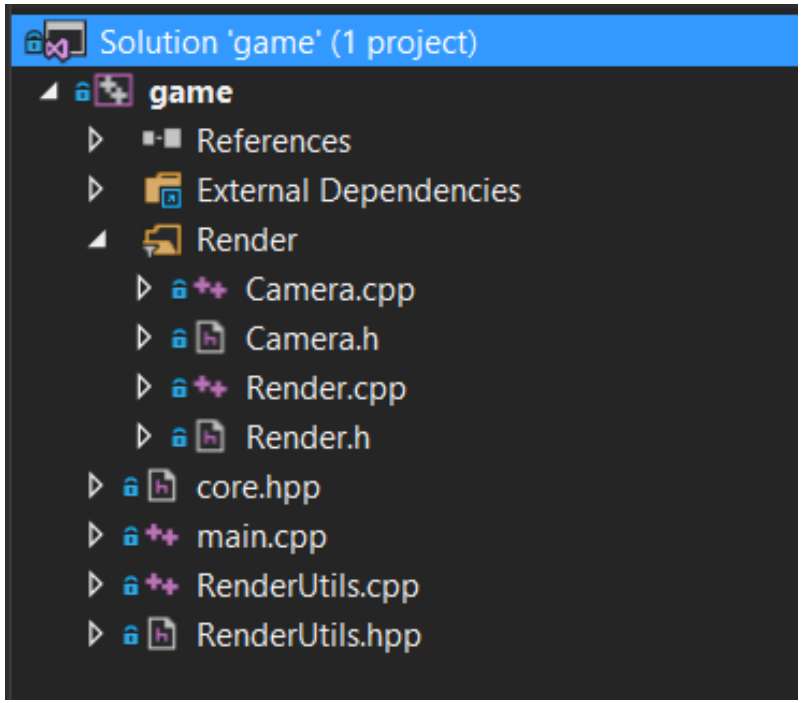
- Contamos con 3 elementos principales
- Carpeta de binarios
- Carpeta de librerías
- Carpeta de esqueleto

- Se proporciona un esqueleto
- Contiene todo lo necesario para empezar a programar

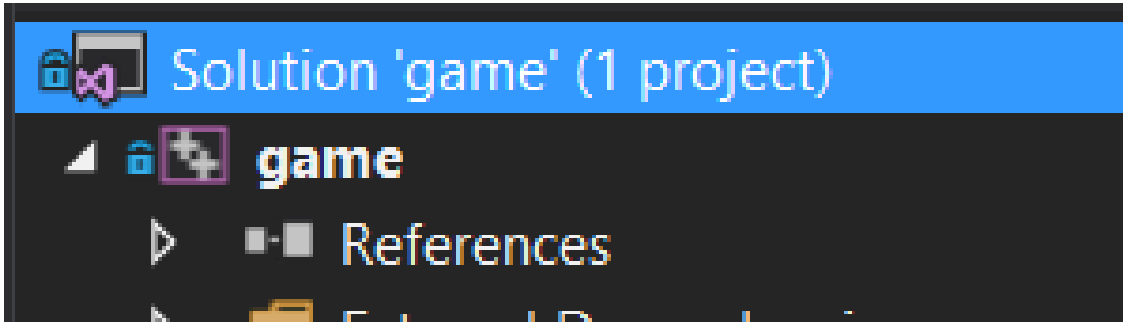


Esqueleto

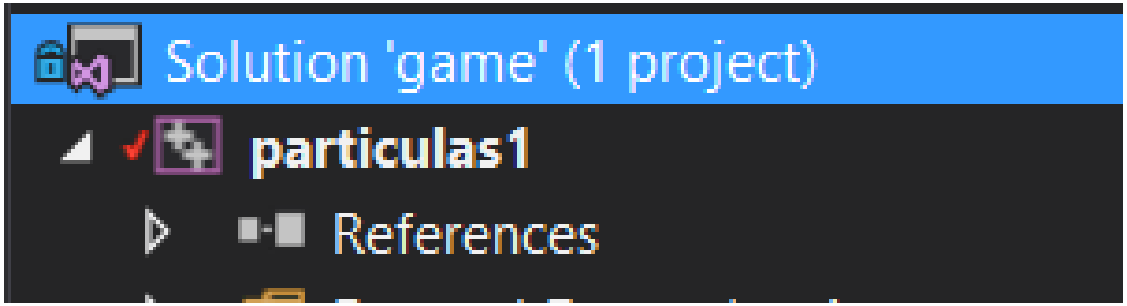
- Se proporciona un esqueleto
- Contiene todo lo necesario para empezar a programar
- Al abrirlo veremos un proyecto de Visual Studio
- Sobre él trabajaremos nuestras prácticas



- En esta carpeta irá el resultado de la compilación
- Generará un .exe con el nombre del proyecto



game.exe



particulas1.exe

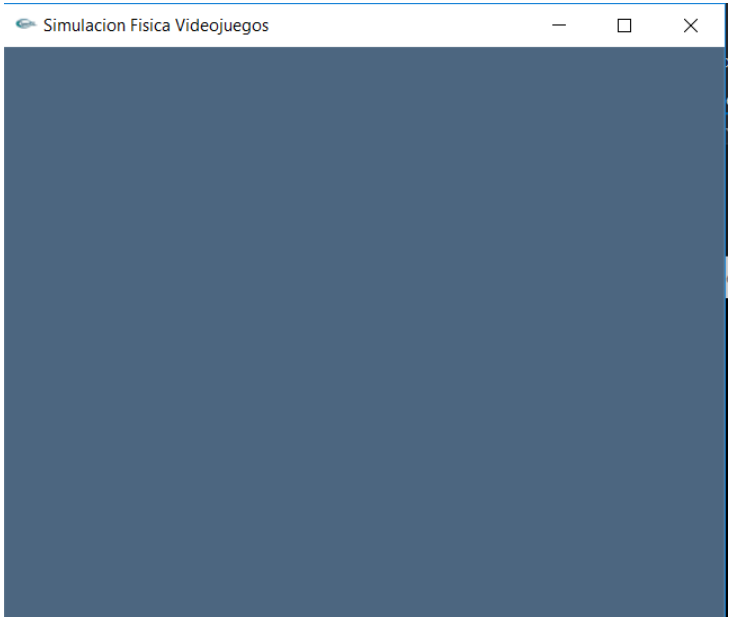
- Se ejecutará desde allí

Librerias

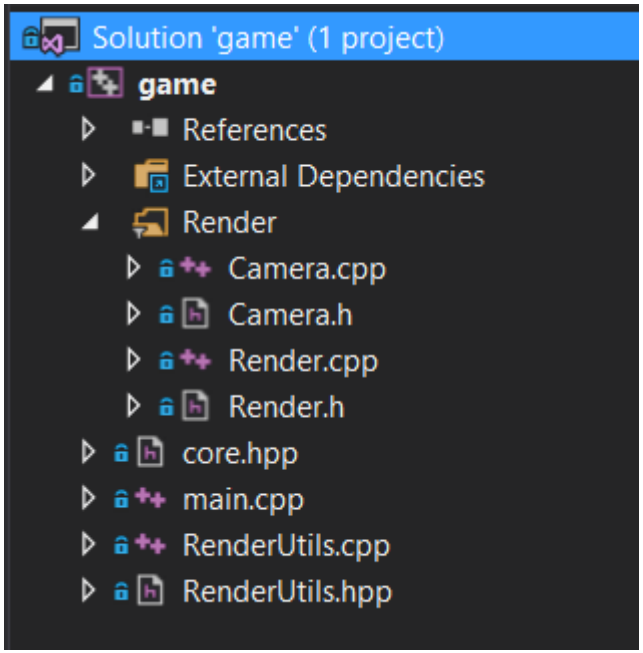
- Al usar ciertos aspectos de PhysX es necesario usar sus librerías
- Todo lo necesario está en la carpeta common
- En los enunciados de las prácticas se indicará cómo usar funcionalidades necesarias
- Para más información
  - https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Index.html
  - https://www.opengl.org/resources/libraries/glut/spec3/spec3.html

- Varias configuraciones
- Debug
  - Para poder encontrar errores y problemas
  - Se pueden ver bien los valores en el Depurador de Visual Studio
- Release
  - Para tener rendimiento en simulaciones costosas
  - Se puede depurar
    - Los valores no son fiables
    - Algunas funciones no existirán → inline

- Si compilamos y ejecutamos
- Obtenemos una ventana renderizando
  - Escena vacía
  - Camara en una posición
  - Posibilidad de mover la cámara interactivamente



- El proyecto viene con una serie de ficheros con funcionalidades básicas
- main.cpp es el punto de partida



initPhysics

- Se llama automáticamente al principio del juego
- Se utiliza para inicializar todos los elementos necesarios para la simulación
- Creación de instancias básicas de PhysX
- Espacio de código para la práctica concreta

```
void initPhysics(bool interactive)
{
   PX_UNUSED(interactive);

   gFoundation = PxCreateFoundation(PX_FOUNDATION_VERSION, gAllocator, gErrorCallback);

   gPvd = PxCreatePvd(*gFoundation);
   PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_HOST, 5425, 10);
   gPvd->connect(*transport,PxPvdInstrumentationFlag::eALL);

   gPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation, PxTolerancesScale(),true,gPvd);

   gMaterial = gPhysics->createMaterial(0.5f, 0.5f, 0.6f);

   // Add custom application code
   // ...
}
```

- Función que se llamará para cada paso de simulación
  - Frame
- Será necesario completarla si necesitamos que nuestra práctica haga algo en cada Frame
- Recibe el tiempo transcurrido desde la última llamada

```
void stepPhysics(bool interactive, double t)
{
   PX_UNUSED(interactive);
   PX_UNUSED(t);

   // Add custom application code
   // ...
}
```

- Llamada automáticamente al terminar la simulación
  - Cerrar el juego
- Limpiar todo aquello que hayamos creado
  - Memoria

```
void cleanupPhysics(bool interactive)
{
    PX_UNUSED(interactive);

    // Add custom application code
    // ...

    gPhysics->release();
    PxPvdTransport* transport = gPvd->getTransport();
    gPvd->release();
    transport->release();

    gFoundation->release();
}
```

- Todas las funciones y clases de PhysX usan un namespace
  - physx
- No suele haber problema en pasarlos al root
  - using namespace physx
- Algunas configuraciones básicas en core.hpp
- No está puesto ahí, pero es posible añadirlo a los .cpp

Vectores

- Usaremos las clases de vectores de PhysX
- En core.hpp hay un typedef a una clase Vector3
  - Así podríamos cambiarlo y usar alguna otra si se necesitara
- Cuenta con las operaciones normales
  - Varias formas de creación
  - Asignación
  - Comparación (comparación de float)
  - Suma (+, +=)
  - Resta (-, -=)
  - Multiplicación por escalar (*, *=)
  - División por escalar (/, /=)
  - Dar la vuelta (-)

Vectores

- Longitud (magnitude, magnitudeSquared)
- Normalizar (normalize)
  - Deja al vector con longitud 1.0
  - Devuelve la longitud que tenía el vector
- Calcular normalizado (getNormalized)
  - Devuelve un vector con longitud 1.0
  - No toca el vector
- Producto escalar (dot)
- Producto vectorial (cross)

- El sistema de pintado está ya programado
- Basado en OpenGL y glut
- El centro es una clase llamada RenderItem
- Realiza toda la funcionalidad necesaria
- Podéis extenderla con lo que necesitéis
  - Pero no es imprescindible tocarla
  - Si falta algo para hacer la práctica hablad con un profesor

RenderItem

- El sistema de pintado está encapsulado
- Clase que se utiliza para registrar algo para pintar
- Conteo de referencias
  - Cuando llega a cero se libera a sí mismo
- Contiene
  - Shape con la geometría a pintar
  - Transform para saber su posición y orientación
  - Color

```
class RenderItem
{
public:
    RenderItem(physx::PxShape* _shape, const physx::PxTransform* _trans,
Vector3 _color);
```

RenderItem: Shape

- Para crear un shape existe una función en RenderUtils.hpp

```
PxShape* CreateShape(const PxGeometry& geo);
```

- Usaremos las geometrías de PhysX
  - Todas derivan de una clase general PxGeometry
  - PxSphereGeometry
  - PxBoxGeometry
  - PxCapsuleGeometry
- Añadir un shape a un RenderItem añade una referencia al shape
- Si no guardamos una referencia es necesario llamar a la función release del shape

RenderItem: Transform

- Es un puntero a una instancia Transform
- Es una clase de PhysX
- Encapsula una posición, rotación

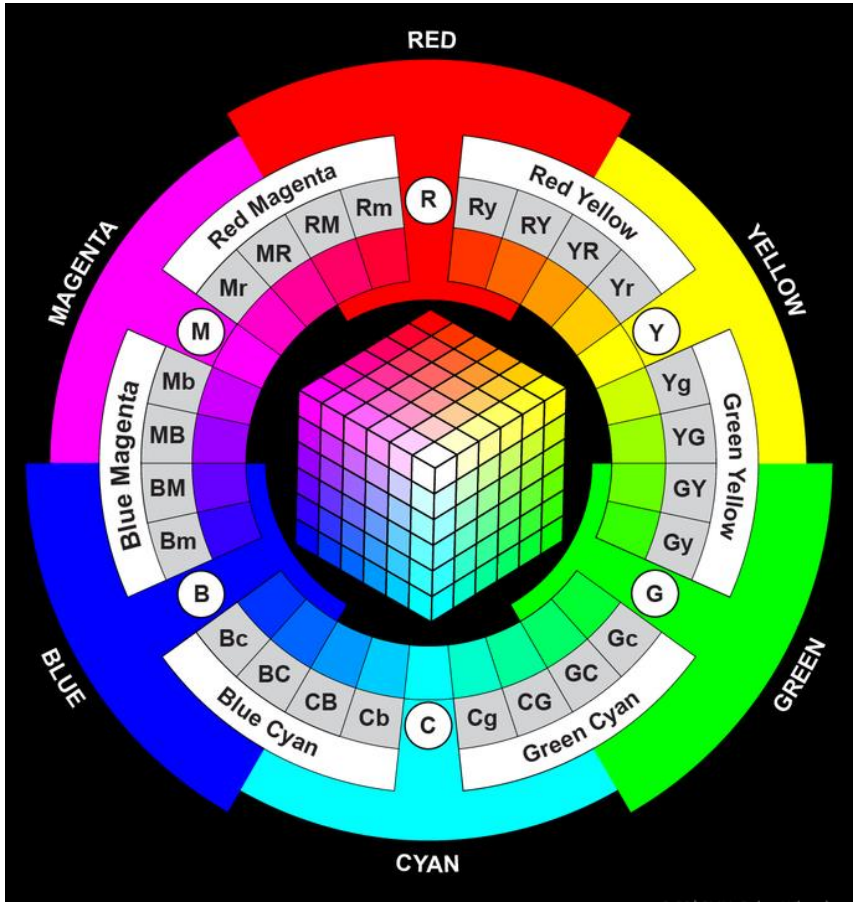
```
PxTransform(const PxVec3& position) : q(PxIdentity), p(position)

PxTransform(const PxQuat& orientation) : q(orientation), p(0)

PxTransform(const PxVec3& p0, const PxQuat& q0) : q(q0), p(p0)
```

- Puntero ya que cuando algo se mueva lo tendremos que ir actualizando
- El código del juego guarda una instancia a Transform
- RenderItem guarda el puntero a esa instancia
- Siempre se pinta en la posición actualizada

- Vector3 con los valores de R, G, B



- Función para obtener la cámara en RenderUtils.hpp

```
Camera* GetCamera()
```

- A partir de ella se pueden obtener
- Posición del ojo → getEye
- Dirección en la que mira → getDir
  - Normalizada
- Se mueve interactivamente con el ratón y el teclado
  - A, W, S, D
- Se puede también mover mediante código si lo necesitáis

Tiempo

- El tiempo se cuenta en milisegundos
- Se puede consultar el tiempo actual en cualquier momento
- Es un número que siempre va aumentando

double GetLastTime()

- Se actualiza sólo cada frame
- Si han pasado 3 frames y el juego va a 30fps valdrá 100
  - 100 milisegundos

Teclas

- Existe la opción de realizar acciones como respuesta a pulsaciones de teclas
- Hay un callback que se llama cada vez que se pulsa una tecla

```
void keyPress(unsigned char key, const PxTransform& camera)
```

- La tenéis definida en main.cpp

Variables globales

- En las prácticas será necesario guardar instancias de objetos
- En un juego completo seguramente existirían gestores de esas instancias
  - Clases para su almacenaje y gestión de ciclo de vida
- En nuestras prácticas no es necesario
- Una simple variable global es aceptable
  - Aunque en general no se recomiendan

Entregas

- Las prácticas tendrán fecha de entrega
- Normalmente dos semanas después de realizar el laboratorio
- No es posible recuperar las prácticas
- Se corregirán en clase pasado el plazo de entrega

Entregas

- Normalmente constarán de varios apartados
- Se deben entregar por separado
- Subir un zip contiendo los resultados de la práctica
- El nombre del zip tendrá este formato
  - <nombre_alumno>_<apellido_alumno>_<nombre_practica>.zip
  - Por ejemplo JoseAngel_Garcia_particulas.zip
- El nombre de la práctica será el título que aparezca en el enunciado

Entregas

- Dentro de ese zip habrá una carpeta por apartado con el nombre “practica1”, “practica2”, etc.
- Dentro de la carpeta habrá un .exe resultado de la compilación
- Así como todo el código fuente u otros documentos necesarios para producir y ejecutar ese .exe
  - Excluyendo las librerías proporcionadas en la asignatura