



Entorno de prácticas. Hola mundo. Práctica 0

David Alejo

Raúl Fernández

Lía García



- 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



- Al usar ciertos aspectos de PhysX es necesario usar sus librerías
- En los enunciados de las prácticas se indicará cómo usar funcionalidades necesarias
- Para más información:
 - [PhysX User's Guide](#)
 - [OpenGL Utility Toolkit \(GLUT\)](#)



- <https://github.com/UCM-237/SimulacionFisicaVideojuegos>
 - Contiene todo lo necesario para comenzar a programar
- También se necesita (ver README del repositorio):
 - Carpeta de binarios
 - Carpeta de librerías



- ☐ Las entregas se realizarán mediante git
- ☐ Cada estudiante tendrá un fork al repositorio de la asignatura donde irá subiendo el código de las prácticas
- ☐ La entrega se realizará etiquetando la versión con una etiqueta. El número de versión será:
 - ☐ V0.P.A (P número de la práctica y A apartado)
 - ☐ Por ejemplo, el apartado 3 de la práctica 1 se entregará como V0.1.3
- ☐ Las prácticas son INDIVIDUALES y se corrigen individualmente, de manera oral a cada estudiante. Durante la corrección os podemos pedir que cambiéis código o que nos respondáis a cuestiones de física o programación relacionadas con la práctica



Primera parte: Hola Mundo



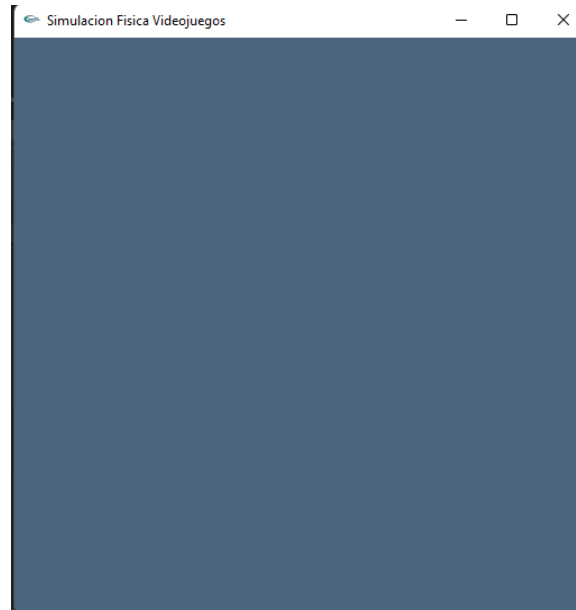
Actividad 1

- Hacer un fork del repositorio de la asignatura
- Generar un proyecto de Visual Studio mediante la opción clonar repositorio
- Descargar las carpetas bin y common (según instrucciones del Readme del repositorio)



Actividad 1: Entorno de prácticas

Hola Mundo





Actividad 1: Entorno de prácticas

Solución "game" (1 de 1 proyecto)

- game
 - Referencias
 - Dependencias externas
 - Render
 - Camera.cpp
 - Camera.h
 - Render.cpp
 - Render.h
 - callbacks.cpp
 - callbacks.hpp
 - core.hpp
 - main.cpp
 - RenderUtils.cpp
 - RenderUtils.hpp



Actividad 1: Entorno de prácticas

- ❑ El main.cpp es el punto de partida
- ❑ Tenemos tres funciones básicas:

```
32
33     // Initialize physics engine
34     +void initPhysics(bool interactive) { ... }
57
58
59     -// Function to configure what happens in each step of physics
60     | // interactive: true if the game is rendering, false if it offline
61     | // t: time passed since last call in milliseconds
62     +void stepPhysics(bool interactive, double t) { ... }
69
70     -// Function to clean data
71     | // Add custom code to the beginning of the function
72     +void cleanupPhysics(bool interactive) { ... }
87
```



- ❑ Se llama automáticamente al principio del juego
- ❑ Se utiliza para inicializar todos los elementos necesarios para la simulación
- ❑ Creación de instancias básicas de PhysX
- ❑ Este es el lugar para el código de inicialización

```
33 // Initialize physics engine
34 void initPhysics(bool interactive)
35 {
36     PX_UNUSED(interactive);
37
38     gFoundation = PxCreateFoundation(PX_FOUNDATION_VERSION, gAllocator, gErrorCallback);
39
40     gPvd = PxCreatePvd(*gFoundation);
41     PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_HOST, 5425, 10);
42     gPvd->connect(*transport, PxPvdInstrumentationFlag::eALL);
43
44     gPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation, PxTolerancesScale(), true, gPvd);
45
46     gMaterial = gPhysics->createMaterial(0.5f, 0.5f, 0.6f);
47
48     // For Solid Rigids ++++++
49     PxSceneDesc sceneDesc(gPhysics->getTolerancesScale());
50     sceneDesc.gravity = PxVec3(0.0f, -9.8f, 0.0f);
51     gDispatcher = PxDefaultCpuDispatcherCreate(2);
52     sceneDesc.cpuDispatcher = gDispatcher;
53     sceneDesc.filterShader = contactReportFilterShader;
54     sceneDesc.simulationEventCallback = &gContactReportCallback;
55     gScene = gPhysics->createScene(sceneDesc);
56 }
57
```



- ❑ Función que se llamará para cada paso de simulación
- ❑ Será necesario completarla si necesitamos que nuestra práctica haga algo en cada Frame (actualice posiciones, por ejemplo)
- ❑ Recibe el tiempo transcurrido desde la última llamada

```
58
59  // Function to configure what happens in each step of physics
60  // interactive: true if the game is rendering, false if it offline
61  // t: time passed since last call in milliseconds
62  void stepPhysics(bool interactive, double t)
63  {
64      PX_UNUSED(interactive);
65      //
66      gScene->simulate(t);
67      gScene->fetchResults(true);
68  }
69
```



- ❑ Llamada automáticamente al terminar la simulación al cerrar el juego
- ❑ Limpiar todo aquello que hayamos creado

```
69
70 // Function to clean data
71 // Add custom code to the begining of the function
72 void cleanupPhysics(bool interactive)
73 {
74     PX_UNUSED(interactive);
75
76     // Rigid Body ++++++
77     gScene->release();
78     gDispatcher->release();
79     // -----
80     gPhysics->release();
81     PxPvdTransport* transport = gPvd->getTransport();
82     gPvd->release();
83     transport->release();
84
85     gFoundation->release();
86 }
87
```



- Además tenemos esta función para interaccionar con el teclado, que podemos completar.

```
88 // Function called when a key is pressed
89 void keyPressed(unsigned char key, const PxTransform& camera)
90 {
91     PX_UNUSED(camera);
92
93     switch(toupper(key))
94     {
95         //case 'B': break;
96         //case ' ': break;
97         case ' ':
98         {
99             break;
100         }
101         default:
102             break;
103     }
104 }
105
```



- ❑ Todas las funciones y clases de PhysX usan un namespace physx
- ❑ using namespace physx
- ❑ Algunas configuraciones básicas en core.hpp



Parte 2: pintar



Geometría

- `PxSphereGeometry`
- `PxBoxGeometry`
- `PxCapsuleGeometry`
- `PxPlaneGeometry`



Crear una forma a partir de esa geometría

- `CreateShape`
 - `PxGeometry`
 - `PxTransform`
 - `Color`



Crear un item para renderizar

- `RenderItem`



- ☐ El sistema de pintado está ya programado
- ☐ Basado en OpenGL y glut
- ☐ El centro es una clase llamada RenderItem (Definida en **RenderUtils**)
- ☐ 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



- ☐ El sistema de pintado está encapsulado
- ☐ RenderItem es la 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



RenderUtils.hpp:

```
1  #ifndef __RENDER_UTILS_H__
2  #define __RENDER_UTILS_H__
3
4  #include "PxPhysicsAPI.h"
5  #include "core.hpp"
6
7  class RenderItem;
8  void RegisterRenderItem(const RenderItem* _item);
9  void DeregisterRenderItem(const RenderItem* _item);
10
11  class RenderItem { ... };
12
61
62  double GetLastTime();
63  Camera* GetCamera();
64
65  physx::PxShape* CreateShape(const physx::PxGeometry& geo);
66
67  #endif
```



- ❑ El primer atributo de RenderItem es **shape** es un puntero a instancia shape de PhysX
- ❑ Usaremos geometrías de Physx. Todas derivan de una clase general PxGeometry
 - ❑ PXSphereGeometry
 - ❑ PxBBoxGeometry
 - ❑ PxCapsuleGeometry
- ❑ Añadir un shape a un RenderItem añade una referencia al shape
- ❑ Para borrar, si no guardamos una referencia es necesario llamar a la función release del shape



- ❑ El atributo **transform** es un puntero a una instancia Transform
- ❑ Es una clase de PhysX
- ❑ Encapsula una posición, rotación
- ❑ 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

```
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)
```



- ☐ El último atributo es **color**
- ☐ Vector4 {R,G,B, alpha}
- ☐ Formato RGB con valores entre 0 y 1
- ☐ Alpha transparencia.
 - ☐ 1 objeto sólido
 - ☐ 0 objeto transparente



Actividad 2

- Pintar una esfera en la posición (0,0,0)



Parte 3: clase Vector3D



- ❑ PhysX tiene implementada una clase de Vectores 3D
- ❑ PERO por motivos didácticos vamos a crear la nuestra de manera que podamos hacer todas las operaciones que necesitemos usando vectores en el espacio



Actividad 3

- Programa tu clase Vector3D para trabajar con comodidad con vectores en el espacio de 3 dimensiones
- Al menos deberá contar con métodos para:
 - Normalizar el vector y obtener su módulo
 - Realizar el producto escalar de ese vector por otro
 - Multiplicar el vector por un escalar
 - Sobrecargar los operadores $=$, $+$, $-$, $*$
- Usa tu clase Vector3D para visualizar en pantalla los ejes de coordenadas



Actividad 3

