GAME ENGINE DEVELOPMENT (/BLOG/?CATEGORY=GAME+ENGINE+DEVELOPMENT)

How does a Game Engine work? An Overview (/blog/how-do-i-build-a-game-engine)

JANUARY 9, 2016 (/BLOG/HOW-DO-I-BUILD-A-GAME-ENGINE) HAROLD SERRANO (/BLOG/?AUTHOR=5485153FE4B024101FA8139B)

Not long ago, I started developing a game engine. At the time, I had limited knowledge on C++ and no knowledge on OpenGL. With all honesty, I was the least qualified person to start such project. Nonetheless, I decided to dive in.

During the first eight months, I read a lot on OpenGL and C++. The first prototypes of the engine were failures. I couldn't even get OpenGL to work right. But I kept going and soon everything started to make sense.

Developing a game engine is not an easy task. It requires knowledge on 3D mathematics, programming and computer graphics. But more than anything, it requires perseverance. So if you promise not to quit, I would like to give you a summary on how to develop a game engine.

A game engine consists of three smaller engines:

- Math Engine
- Rendering Engine
- Physics Engine

The interaction between these engines allows a character to move, jump, collide, etc. It also allows a game scene to produce shadows or a scenery.

The Math Engine

Your first task is to develop a math engine. The math engine handles all linear algebra operations and geometric operations. As a start, you should focus on implementing only vectors and matrix operations. Vectors and Matrices allows a character to translate and rotate, respectively.

Your math engine should contain these vector operations:

- Vector addition & subtraction: Allows a character to move
- Dot Product: Determines how much a vector influences another
- Cross Product: Allows for the creation of a third vector

Your math engine should also contain these matrix operations:

- Transformation
- Transpose
- Inverse
- Identity

The transformation operation allows a character to rotate.

You may want to include Quaternion and Dual-Quaternion operations in the math engine. Quaternions rotate game entities but uses less space and are faster than matrices. Dual-Quaternions offer the beauty of translation and rotation in a single math entity.

The Rendering Engine

To render a pixel on a screen you need to communicate with the GPU. To do so, you need a medium. This medium is called OpenGL. OpenGL is an API and its main purpose is to send data from the CPU to the GPU.

Your engine will render different type of objects. It will render 3D models, images, skymaps, text, sprites, etc. Thus, it is a good idea to set up a Rendering Manager. The Rendering manager will take care of all OpenGL operations.

The rendering manager extracts rendering information from each object. This information is then passed to the GPU through OpenGL buffers. Once in the GPU, the rendering information is processed by OpenGL Shaders.

Normally, each game object contains the following rendering information:

- Vertex Position
- UV Coordinates
- Normals
- Textures

The GPU uses the *vertex position* to assemble the geometry of the object. It uses the *normal data* for lighting operations. And it uses the *UV coordinates* and *texture* to apply images to the object.

Scenegraph

A game will have many game characters. To keep track of all game entities, you need to set up an Entity Manager. The Entity Manager keeps track of all entities that are active in a game.

The entity manager stores each game entity in a container. The best container to use is a scenegraph. Scenegraphs are generic trees and they provide a fast way to traverse game entities. You may want to use C++ vector containers instead. This is all right. The problem is that they are too slow to traverse. Thus, if you can, use scenegraphs instead.

The Engine Loop

A master loop is set up in the engine to continuously render and update the coordinate space of each entity. Some developers call this a Game Loop. I call it an Engine Loop.

The entity manager provides game entities to the Engine Loop. The engine loop then renders each entity by calling the Rendering Engine. It also updates the coordinate space of each entity by calling the Math Engine.

The Update function sets the space transformation of each entity. Vectors and matrices translate and rotate the entity.

The Render function calls the Rendering Manager. The Rendering Manager takes care of rendering the different game entities. For example, 3D characters, sprites, skyboxes, etc.

The Physics Engine

The cool part of a game engine is the Physics engine. But this is also the most complicated. The physics engine determines the position and velocity of an entity. It does this by integrating the external forces acting on the entity.

The most common external force acting on an entity is gravity. By integrating the force of gravity we get the velocity and position of the entity. This information is then used to provide the illusion that the entity is falling. In a nutshell, the physics engine responsibility is to integrate the equation of motion.

The most common algorithms used to integrate the equation of motion are:

- Euler Method
- Verlet Method
- Runge-Kutta Method

The Euler Method is simple to implement, but is the least exact. The Runge-Kutta Method is the method I prefer. It is a bit more complicated, but is also the most precise.

Conclusion

There is a lot more to a game engine. For example, I did not talk about Collision Detection. But, this overview should give you a taste for what you need to implement. Like I said, implementing a game engine is a huge task. Nonetheless, it is a lot of fun. If you like math, coding and are a bit of a nerd, you will love doing this.

PS. Sign up to my newsletter and get Game Engine development tips.

TAGGED: GAME ENGINE (/BLOG/?TAG=GAME+ENGINE)

❤

How does a Rendering Engine work? An overview (/blog/how-to-develop-a-rendering-engine-overview)

How do I start planning my first game using OpenGL and C++?

BOOKS (/MARKETPLACE/) PROJECTS SUPPORT (/SUPPORT/)

CONTACT ME (/CONTACT/)

Copyright © 2015-2018 Harold Serrano