Hogeschool van Amsterdam

Game Technology

Toon Defense

Christian Aguilera (cristian64@gmail.com)

March 27th, 2012

Index

Introduction

Toon Defense is a tower defense game with spaceships and different kind of towers. It is written in plain XNA 4.0 that doesn't use any game engine. The thing that makes this game unique is the visual aspect: 3D objects + cel-shading. Although the game only works in Windows, it is entirely designed to work in Windows Phone with a touch pad; every action performed with the mouse could be taken on by a single finger.

The game consists of creating towers that shoot spaceships. Those spaceships try to complete a path and, if they do, they take one of your lives.

This report includes the most valuable information that I could gathered during the development. Several topics are really interesting and will be explained with certain details; others will be briefly mentioned in these 7 pages.

Beginner with XNA

Before I started this project, I had no real experience with XNA. I did have some experience programming games and C# but, when it comes to XNA, it was rather no experience.

Although I have dealt with SDL and OpenGL, XNA was something new. The way XNA works with the game loop is different. Programmers don't have a real access to the game loop, but they can include code in the Update and Draw methods, which would be executed in every iteration of the loop. This way is definitely better, but it takes some time to get used to it.

With XNA everything is quite easy. To load content such as images, sounds, fonts, effects or models you only need to drag and drop. And, also, to compile and distribute the application is simple; only .NET 4.0 and XNA 4.0 are needed.

3D Modeling

When I decided to create a tower defense in 3D, I didn't know it would be so complicated. Most models have been gotten from www.turbosquid.com or similar sites. However, it's hard to find the model you're looking for. Almost every time I had to modify the model in different ways to make it suit the game. For example, to reduce the number of polygons, to generate an UV mapping for the texture, to scale or rotate, change format, etc. Using Blender I could export models into a particular format that XNA can load (.fbx).

I had barely modeled before, so that my skills at modeling are not that good. And yet, I could manage with Blender. Actually, I have also modeled some spaceships and towers for the game myself –the simplest ones–. Blender let me do this.

Cel-Shading Effects

In my opinion, the cel-shading effect in the game is the coolest thing. It really makes the game beautiful. To make it possible I looked for XNA effects on the Internet. After three or four tries, I found the one the game uses. Only a few parameters had to be changed.

The first effect I tried had no outline, so objects didn't look good. The next one I found did have an outline that was created after projecting the object on a transparent render target and dilating the projected image. This effect wasn't good because the Z-buffer test was broken; after rendering the image to generate the outline, the object stops being a 3D-object to become a sprite. Finally, I found the one that game uses. In this game, the outline is generating by scaling the model and rendering the model again with a black texture. In this case, the object stays as a 3D-object.

Particle System

At the beginning, I thought I would program my own particle system. It is not complicated as only a list of sprites with texture is needed. In this list you store particles with a certain life that die after some time. This is the easy way, where particles are only sprites and no 3D-objects.

However, there are already some XNA examples on the Internet dealing with particles systems. The example I found really works pretty well. It works with squares in the 3D world. Those squares are directed to the camera, so they look like sprites. What it more is that those square, as 3D-objects, can be behind other objects; it wouldn't be possible with sprites. The feeling is really, really good. It is also efficient because the list of particles includes vertices, so every particle can be drawn with the same primitive call.

I only had to add this particle system and make it work with my camera. I adjusted some parameters for the existing particles and, then, I also defined more kinds of particles such as vortex, plasma, bubble, electricity.

Mathematical Issues

I had to face many mathematical issues. Mainly I had to calculate rotations. Every spaceship has to rotate in the Y component as they fly. Others have to rotate in the X component or Z component, depending on the spaceship. And some towers need to aim to the target, so they need to rotate in the Y component, too.

Missiles are also incredibly complicated. They need to rotate in all three axis at the same time. It took me many hours until I got something that looks great. However, it could still be better, since some times, when the target of the missile is just above, the missile doesn't rotate properly. The missile also needed to move real. That is, they have to be launched, draw a parabola and hit the target. If you the speed was too high, it would never hit the target –as the Moon doesn't hit the Earth–. And it can't move slower than the target, otherwise it would never hit it, either.

Performance

It was surprising that the game runs pretty fast in every machine I tried. One could expect it to work very slow when 100 objects are on the screen. However, it works pretty well even in netbooks. In lame computers, anti-aliasing has to be disable, but still the game looks great because of the cel-shading effect.

3D Picking

Another important issue is the 3D picking. To select towers or spaceships, it's needed to convert mouse coordinates into 3D positions. When the player clicks the screen, a ray is traced from the camera to the 3D world. This ray is used to find intersections with models, that is, intersections with every triangle of every model. Knowing this, it's possible to know what object the player clicked on.

It's possible to obtain this ray with Viewport.Project, given a screen position. This is also need while creating towers. In this case, the intersection of the ray is checked with the plane XZ.

There is also another function used: Viewport.Unproject. Sometimes, 2D images have to be drawn on the screen, close to 3D objects. For example, the life bar of spaceships are 2D images. It is needed to convert 3D positions into screen positions, in order to draw sprites just there. With Unproject we can convert 3D positions into a (x, y) point on the screen.

GUI Library

This is something I really did wrong. It doesn't mean I had another chance, because I didn't want to program my own GUI library. I spent some time on searching the web to find a simple library, and it turned out that there is nothing useful for XNA 4.0 –or I didn't find it–.

At some point, I decided not to use anything, and just draw buttons and sprites and check if the mouse is just on the sprite I drew. It was simple at the beginning, but it got a lot worse. After a few buttons –moving buttons–, labels, icons, etc., it got awful to deal with the code that manage those objects.

Next time I have to do something like this, either I will find something or I'll write my own simple GUI library for buttons, checkboxes and some common functions.

Sound Effects

I had in mind a lot of spectacular sounds for missiles, explosions, lasers, fire, hits, etc. However, after a few towers were built, it got very annoying to play. The game was noisy, so I had to add a button to disable sounds. Eventually, I also reduce the volume of most of the sounds, so only ambient music and other sounds to create or upgrade towers remain. This is the only way sounds can exist in such a kind of game. When the game goes faster (up to 4x), the OST also speeds up.

Balancing

Definitely, this is the hardest task of the game. Balancing a tower defense is quite complicated. Every little change you make, affects the whole balance. If after writing 50 rounds, you realize some tower is too powerful, you need to rewrite those 50 rounds.

Right now the game contains more than 100 rounds and it's pretty fun to play for 1h or so. I need to play a lot of hours to get to know when the game is easy and when is hard. I also have to persuade a few friends to play the game. Sometimes I needed to blackmail them, since they didn't want to play anymore; neither did I.

I guess, some maths are needed here to balance the game really well to make it fun. For example, the game should avoid this kind of problems:

- Player never has too much money. At least, not significantly more than the most expensive tower. If you do, that means the game, or a part of the game, is too easy.
- Player can't go more than 3-4 waves without building or upgrading a tower and without losing a life. If he or she can, that means you've got a part of the game that is too easy.
- Every tower has its use. There shouldn't be towers that are not useful in any valid strategy the player uses.
- Player can't beat the game on the first try on the hardest difficulty. Maybe some very experienced players can get lucky and do it, but the fun part of a TD game is figuring out which towers work when and where to place them. If you just finish the game on the first try that takes all the fun out.