Senior-Project Final-Paper
The Game Creation Process

By: Steven G. Peterson

Submitted To:
The Faculty and Staff of:
The Department of Computer Science
Rowan University

December 7th, 2005

Contents

I Introduction

I.1 E3 – The Adventure Begins

Having decided I wanted to work in the video-game development industry after graduating from Rowan, I traveled out to the L.A. Convention Center for the E3 (Electronic Entertainment Expo) convention. When I returned I had a rather long to-do list for the upcoming semester. Essentially if I wanted anyone to even consider me I need to do several things:

- Compile a portfolio of my projects to date.
- Complete a game-related tech demo that shows off my talents.
- Revamp my resume (again), to make it focused and industry related.
- Put it all on a web-site, where people can find it.

Through numerous conversations with producers and lead-programmers it became clear, that without a solid demo fronting a well rounded portfolio, the resume was worthless.[1] This then would be the goal of the "Senior Project" I had to complete during my last semester (Fall, 2005).

The next most important thing I took away from E3, besides a large stack of business cards was the one word "*Torque*". I wasn't yet sure what it meant, but it appeared I was destined to find out! Luckily for Google and a kiosk touting free internet access this was *not* a life long quest.

Considering the possibility of making a game for senior project, I'd been searching for a 3D game-engine for 6 months already. I needed cheap, dirt-cheap in software terms, but didn't want to be wasting my time either. After the dozens of product-pages and hundreds of screen-shots I'd seen, I knew in less than ten minutes that the *Torque-Game-Engine* was what I'd been searching for.

I.2 Original Concept and Design

Back on the streets of Glassboro(Rowan), and having decided to develop a game for Senior-Project, the next question was what game? Since I only had three months, some steps would have to be cut out. Instead of building a game from scratch, I would take a new approach on an old game.

The idea was to fully re-implement Nintendo's original The Legend of Zelda game from 1987 in a fully 3-Dimensional format using a modern game engine. New features would hopefully include complete environmental effects (sun rise/set, weather etc.) for a more immersive experience, a deeper and more animated combat system, and if time permitted, a "third quest" completely designed by me.

1 Looking back, we did *lots* of projects at Rowan, however, compiling a portfolio was never *once* mentioned in my 4.5 years there. My job search has revealed that, as a programmer, having a solid portfolio is equally *as* or *more* important than any artist, or writer. If I were asked "what's the one place the Rowan C.S. Dept. could improve?", I would say, to emphasize putting all of these projects together into a presentable portfolio on the web! (That and buy a new building, but some things are easier achieved than others...)

I hate to say “imagine action-game 'A' meets adventure-game 'B' except with a puzzle-RPG twist...” but then no-one is without their influences. I had to bring Link into 3-Dimensions, *stat,* and I wasn't about to re-invent the wheel.

One of the things that made The Legend of Zelda unique was that, even though it had a linear story line, it had completely non-linear game play. Something difficult to achieve even in today's games. I decided the best approach was a 3rd-person free-roam system in a completely simulated world, ala “Grand-Theft-Auto: San Andreas”.

Combat and puzzles in the original game were also fairly simplified for their 2D presentation. I wanted to give Link the combat-expertise and puzzle-solving dexterity of the Prince of Persia.

I.3 Tools and Development/Target Environments

The list of tools to make a game is actually fairly long. A game engine defines the virtual world a game takes place in but gives u little to no power to create the objects that populate the world. Character models, buildings, plants, items, weapons, and other objects all have to be modeled, skinned, textured, animated and then imported into the game world. Then you need IDE's and compilers to manage the code of course.

Some engines come with a mostly complete tool-chain for game-development. Others just list the suggested tools they support. Either way you have to start with a game engine.

I purchased the **Torque Game Engine** made cheaply available by GarageGames to “Independent Game Developers”. It works well with numerous professional and industry-standard tools, as well as their open-source and free equivalents. For me the answers were as follows.

My Tool Box:

- Blender - all (small) objects and character models, and animations
- QuArK - all buildings and 'interiors'
- Gimp - textures, heightmaps, and all my other abusive photo-chopping needs
- Torsion - Torque Scripting Editor and Debugger
- MS-VC6 - for all my C++ compiler needs...

II The Torque Game Engine – An Overview

II.1 What is Torque? Who Makes it? Who Uses It?

As I mentioned before, the core technology for my game-project is the **Torque Game Engine** from GarageGames (www.garagegames.com).[2] Torque is a solid current(soon to be last) generation, AAA game engine originally derived from the "Tribes 2" engine. Since then, it has come great leaps and bounds and has been used to build numerous other commercial games.

In essence, Torque is a 3-D simulation platform. It renders a developer-definable, contained universe, that exists inside a six-side cube known as the sky box. One can freely explore this universe from the point of view of a "camera". If the camera is locked to a player it creates a first-person effect, if it follows a player-character, it creates a 3rd-person effect, if it's high-in the sky you can have a god-view of the terrain below.

By coloring the walls of the sky box, or mapping images to it, one can create colored skies, clouds, star-fields etc. By adjusting the ambient light level inside the box, day and night can be simulated. If one were to turn off the terrain, and set gravity to zero, one could even create a space-sim.

Torque also includes functions for calling object animations, manipulating characters, adding physics to your simulations, adding sounds and music etc. The point is, it gives you the tools to create anything you want (inside the skybox), while imposing as few limitations as possible on your imagination.

The **Torque Game Engine** which is comprised of more than a half-million lines of code, or some 2,200+ classes is available for the bargain-basement price of $100 to the *Independent Developer.*[3] According to the EULA[4] that is "Any individual, making *a game*, whose annual income/sales-revenue is less than $250,000." If one does not fit this mold, or grows out of it, one can purchase or upgrade to the commercial license for around $500.

For those unaware, any other engine in this tier can *only* be licensed commercially. The prices start in the tens-of thousands of dollars and go to several hundred-thousand dollars and the engines are not made available to companies without a 'proven-track record of successful commercial games'.

So for a beginner-hobbyists price one essentially gets a commercial grade product from Torque. Not just hobbyists are using it though. It was recently announced[5] that after evaluating

---

2 It should be noted that GarageGames has produced several spin off products from their core-engine known as the "Torque-Game-Engine". These include the retro "Torque-2D Engine", and the next-gen "Torque Shader Engine" and probably more in the future. However, whenever I refer to *Torque* or *TGE* I mean the original Torque-Game-Engine, specifically versions 1.3, and 1.3.5 – the most recent stable versions available at the time.

3 This would be known as the "Indy-License", a licensing model I believe they pioneered.

4 EULA – End User License Agreement

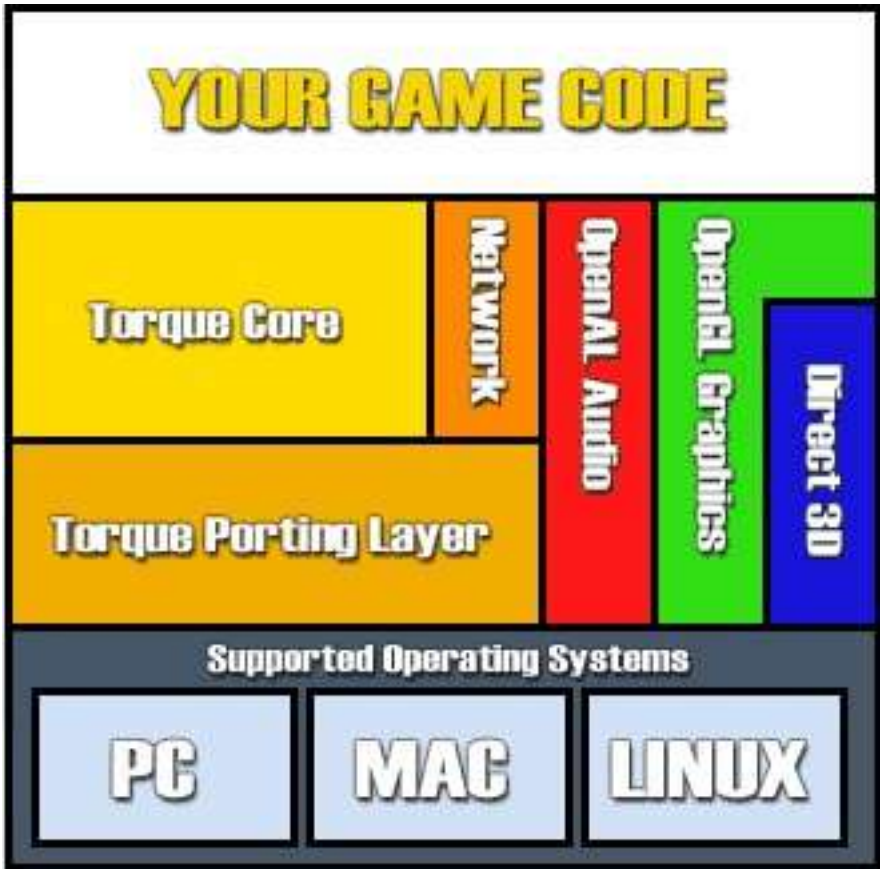
5 http://www.garagegames.com/blogs/34977/9238
http://www.garagegames.com/blogs/34977/9292 (warning, lots of pictures on this one)
http://www.garagegames.com/news/9236

more than 40 game-engines and simulators, NASA has picked Torque to help them do whatever it is that they actually do in between flights to the moon.

Another great thing about working with Torque is that GarageGames hosts an outstanding open-source style community. Even though it's not really “free as in beer” open, once the licensing fee is paid, one can freely download the SDK which includes complete source code, and Doxygen Docs. One also gets access to the complete online-documentation, all of the “private” forums, and a treasure-trove of user-submitted resources, which include hints, tips, tricks, code samples, script modules and large patches.

II.2 What is a Game Engine?

So now that you know more than you ever wanted to about the Torque Game Engine and you have *some* idea what it does your probably still scratching your head trying to figure out what it *is*. Well, I'll tell you and I'll even use my first picture!



(Fig. 1 The Torque Game Engine)

This image is taken from the TGE documentation. The entire middle layer of the diagram would be considered the “game-engine”, and is included in the C++ source provided in the SDK. Most of the effects described above actually occur in the top-left *Torque-Core* section.

As the diagram attempts to show, the larger part of the engine's job is providing a abstraction-layer between the core that the developer actually uses and the OS for cross-platform support. Similar abstraction is provided for sound and graphics services and a strong networking module is built in to the core so it's there when you need it.

This lets the programmer spend as much time as possible in the *core* of the engine, and allows him to use all of those services without worrying how they work or even *if* they'll work depending on the system configuration.

II.3 The Torque Engine in C++

The bulk of the Torque-engine is written in C++.[6] Like I said, you get the complete source code when you buy Torque, to play with as you will. If you look inside the SDK (Software development Kit) there is an "engine" folder, which contains all of the C++, and only C++ with only a few exceptions. Therefore Torque's C++ code is commonly referred to as the "engine code" as opposed to Torque-Script which I'll cover next.

While the vast majority of the C++, one will never have to touch or learn about, if you develop with Torque, you will eventually want to modify something in it. This will of course require a C++ compiler. The Microsoft compilers are supported from V.C.6 and up as is GCC.

II.4 Torque-Scripting Language

While the brunt of the rendering and other CPU intensive processes is handled in the C++, even this is too low-level for a streamlined game-development process. To ostensibly[7] make life easier for everyone involved, most engines include a scripting-language in which a lot of the game can be written, or at least prototyped.

In Fig.1, the *Your Game Code* block represents the *Torque Scripting Language*(TSL) layer. Torque Script is a typeless C++ style language that can be quickly written and edited. It is compiled at run time but the binaries *are* saved, so that if the script does not change it need not be recompiled next time.[8]

Helpful Hint: All Torque-Script files must be *called* from somewhere, for them to be compiled and executed. Since there are no "Include" statements in TSL, the engine includes scripts as they are executed. The best way to do this is to use the *exec("./foo.cs")* command from another script like server/scripts/game.cs.

---

6 For those curious, a little 'lexx' and 'yacc' was used around the edges too, but there generally is no reason to touch it.

7 In the long run use of the script-layer could provide a huge return. Unfortunately most of my changes were made in the *engine code*(C++), or involved passing data between the script and engine layers, so I usually felt like it was working against me.

8 For those keeping score: Torque-Script files have a **.cs** extension and are a plain-text file. The compiled Torque-script binaries are stored in a *new* file with a **.cs.dso** extension that keeps the same filename as the original **.cs** file.

III Building the Environment

III.1 Beginning Zelda – An Overview

Now that your up to speed on the Torque Game Engine and my lofty project goals, it's time to start building. Following is the general process I went through in the approximate order, to make my game demo.

This section should give you an idea of how the game engine is used along with a few 3rd-party tools to start defining an immersive environment.

III.2 The Height Map

There are several ways to define the elevation and contours of the terrain in TGE. One of the easiest is to import *height map*. The height map is a 256x256px image in grayscale mode that portrays an elevation map. Darker areas represent lower elevations (#000000 = minimum elevation), and lighter areas represent higher elevations (#FFFFFF = maximum elevation).

Since I wanted a highly customized terrain-map, I made mine in The GIMP. I started by taking a high-resolution image of the original Zelda map. I cut out all the features I didn't want, like NPC, caves-entrances, trees and bridges. I then converted the image to black and white and scaled it down to the right size. The Zelda map is actually about 250x80 pixels, so I put it on a black background (low-elevation) and the extra space will eventually become ocean.

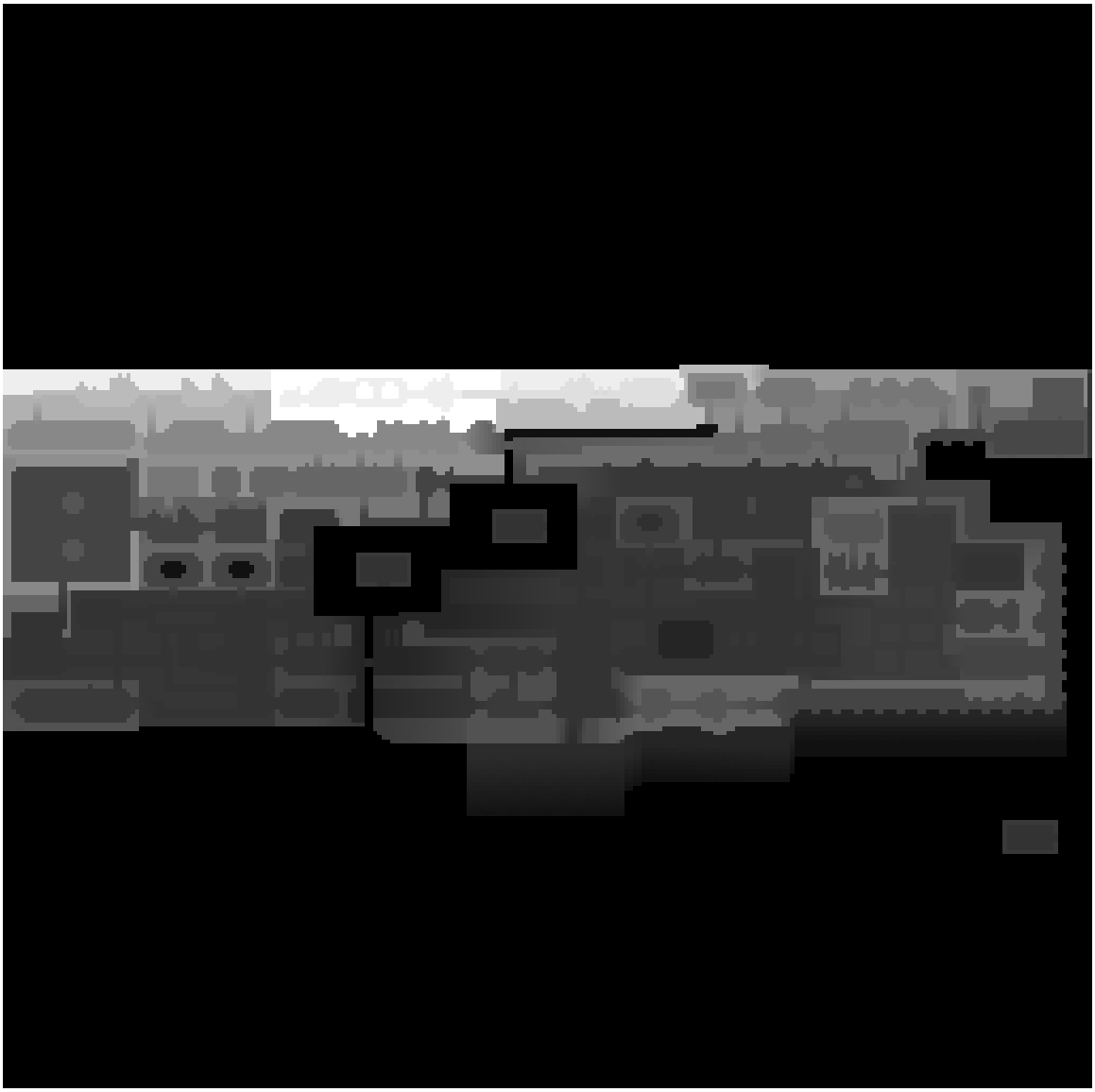
Next, going one region at a time I changed the colors of the entire map to create an elevation map that looks exactly like the original over-world map. To get the elevation differences correct took a little math and a lot of trial and error.

To import the map it has to first be in the right folder in the games directory structure, which happens to be: common\editor\heightScripts. Then it can be imported and applied from the Torque's in-game terrain-editor.

The last thing to do before clicking “apply” is to set the min. and max. elevation values in meters. I used 0 and 300 because I thought that was the maximum range. I later discovered TGE-1.3 supports at least 0-500, it may be more now.

Fig.2. Shows the height map I used. Getting this correct was a tedious process that took many iterations, and it's not finished. It works though.

(Fig.2 Legend of Zelda – Height Map)



III.3 Celestials - Controlling the Heavens

On our quest to create a world, nothing can be taken for granted. In TGE it is easy to put a sun-flare in the sky (fxSunLight). It can even be animated. However it is in no-way related to the sun-vector used for calculating shadows, or the ambient light level, or the color of the sky box. To create a *day/night cycle,* all of these things and more have to be tied together and animated.

The *Celestial Manager* is a patch I got out from the GarageGames user-submitted resources. There's two major parts to the patch:

1. The Celestial.cc class
2. “Hooks” into portions of existing code-base.

The class consists of functions for animating the sun or other celestial body across the sky. Rather than a fixed path across the sky, it uses some spherical geometry to follow an orbit that gently shifts with the seasons. It also calculates proper light intensities, and a color key that will be used later.

The “hooks” are responsible for creating the actual day/night effect. This is where we patch the code through-out the engine, to take into account the values calculated by the celestial class. The most common task is multiplying the intensity and color values of the ambient light shining on any given object by the current intensity and color-key values from the celestial class. This is how objects in the world get not only lighter and darker, but are also shaded with white light in the day, reds and pinks at sunset/sunrise and midnight-blue at night.

This has to be done for basically every imaginable object: terrain, water-surfaces, items, sprites, buildings, trees, clouds, and even the skybox itself. Essentially the more working hooks one can create, the more realistic the day/night cycle becomes. There is nearly infinite possibility for greater detail. However, it eventually becomes a trade-off for performance and development-time.

There are still several unsolved problems in the celestial code I have. While I fixed a number of hooks and even added new ones several are still broken. For example the one's that light the player, trees and buildings. Also, even though this celestial concept could be used to create a sun, moon, battle-station, star-field or anything else it only lets you create *one.*

III.4 Notes on Shadows

Shadows are another thing that has to be accounted for in a day/night cycle, and they become rather tricky. The first problem is if an objects shadow isn't updated frequently enough, then the player will see the shadows jump, particularly at sunrise/sunset. So no cheating allowed. The second problem is that for each object, collision detection has to be done between the light-source, object and ground to find out where the shadow falls. Since collision detection is expensive so is calculating shadows.

Torque solves this problem in several ways. For example, it does calculate real-time shadows for the player. The most notable example though, is *baked-in lighting.* On mission load, shadows are calculated for the terrain and buildings. From that, a static shadow map is created

and overlaid on top of the terrain. This works great as long as the building, terrain, or light source doesn't move.

The current celestial approach is to bypass this whole process. *No Shadows, No Problem.* After all, no terrain shadows are a lot harder to notice than *wrong* terrain shadows. Unfortunately, the lack of shadows tends to give even the brightest of worlds an overcast feel.

One possible solution is to pre-calculate a set of terrain-light maps, then at runtime blend them in and out to prevent any popping effects. This solution isn't without it's problems though. The light-maps are between one and three megabytes, and take several seconds to calculate. Either the first time a person runs the game, the load time will be horrifically slow, or the size of the game-install base becomes dramatically larger.

Even worse, the last time this was tried, there was a small (reportedly 500ms give or take) hiccup when loading a new light map. While this might be acceptable for a long RPG, in an FPS game it would not be. Either way, I'm going to try re-implementing this in the future to see how it works out. Current hardware alone may solve the problem, if not, some creative programing and maybe even an extra thread may solve help to smooth it over. Next-Gen, these types of issues are solved using pixel-shader technology and true multi-threaded applications that will take advantage of the latest feature-filled GPU's and true multi-cored CPU systems.

III.5 GameManager – Date/Time and Dynamic Weather

Not yet content, I found the *GameManager* resource. The GameManager is written entirely in Torque-Script which makes it really easy to learn and work with. It does three things. The first is a top level scheduler function. Basically it schedules itself to be updated every 1000ms. On each iteration, it then loops through a set of sub-modules which specify how often *they* need to be updated, and calls the module if needed. The schedule manager also defines how many real-time seconds equal one *game-hour*.

The second task is a *time-manager*. This very simply implements the concept of a calendar, based on pre-defined inputs for things like days in year, days in month, months in year, etc. It then outputs calendar events, including sunrise, sunset and the date, *based on time*, to the in-game text-box.

The third task is a *dynamic-weather-manager*. This is one of the modules I had the most fun with early on. Basically it tracks it's own number between approx. 960 and 1040 to use a weather-pressure. The number is randomly generated on start-up, and updates itself once per game hour, based on random-dice rolls, with checks to make sure it doesn't' rise or fall too much at once.

Once the weather pressure has been updated, a series of checks is done to see if it crossed any thresholds that would signify a change in weather.

- Below 980 is a thunder-storm.
- 980 to 1000 is rain.
- 1000 to 1020 is cloudy skies.
- Above 1020 is clear skies.

Torque can fade clouds in and out. It can also turn on rain and even fade it in and out based on drops-per-minute. The frequency of lighting strikes can be changed, and their position can be confined to an area of the map where the player actually is. Though I didn't implement it, others have added a check to see if the player was hit by a lightning strike, adding in custom sound effects and player-damage.

When I first installed the weather manager, it did not consider wind velocity, even though all of the in-engine environmental effects use it. Therefore whatever the default was, it remained constant. The clouds always moved the same direction, the rain-storm was always the same rain. This was some of the first changes I personally made to the game code.

First I created a function that would update the wind velocity every game hour based on more random dice rolls. I set the range for wind velocity to be between (-4, -4) and (4, 4). Since there is no fade for a change in wind direction, I couldn't have any major shifts at once. As it is now, it will only change by zero or one points, on one axis every game hour.

The only problem I encountered with this was that weird things would break if the wind velocity was initialized to (0, 0). Some careful programming ensured this case would not occur.

To further customize rainstorms the intensity can be set, mainly through setting the drops-per-minute. Now that I had dynamic wind I decided to calculate a new drops-per-minute, based on the current wind-velocity every time the startRain() routine is called. Essentially I take the absolute value of the wind velocity on each axis, add them together, multiply by a constant like 500, and add another 2000. This gives me a comfortable range between 2000 and 6000 drops per minute.[9]

While wind is continually changing, the rain-intensity is only changed on the creation of a new rainstorm. This seems to create plenty of variance in the rain from storm to storm, and even in a single storm over time. This is a good example of how more detail can always be added, but is not always worth the time, effort, or CPU-cycles.

III.6 Enviro-Torque

While the above two resources were cool they were far from perfect. Both were out-dated by over a year (and a major-release of TGE), in which time other users had posted numerous bug fixes. It didn't seem to make sense that every user should waste their time as I had to figuring out these fixes and merging them into the code.

Worse several people admitted to getting these two resources specifically to work together but no one said how. So I resolved to create a resource and give back to the community before moving on.

9 It occurs to me that in a future version I could alter the speed the drops fall in a similar manner so that some days it's really sheets of rain, while others, it's a slow gentle rain.

III.6.1 Problems with Integration

The biggest issue was that both resources, Celestials and GameManager are functions of time, however one is contained entirely in C++ and the other in script. To integrate them(using the term loosely) I followed a simple strategy. All of the input-values that the GameManager components needed that *weren't* already contained in the Celestial class I added there any ways. Then I made all of the inputs to Celestial, that would have to be shared, visible to Torque Script, making any needed conversions along the way. Last I altered GameManager to pull these inputs from the celestial class in the engine, rather than from where they were previously defined in the scripts. This approach left a lot to be desired but seems to have been a good quick solution.

The last thing I did was track down a resource that generates random-clouds based on a fractal to create amazing skies. Normally Torque scrolls a .bmp image of clouds on “cloud-layers” inside the sky box. By using several layers, moving at different speeds the effects are quite good. The fractal-skies simply replaces the bmp image.

Now I already had decent storm-clouds, courtesy of the Torque Demo, but when the weather-manager turned them off, the flat blue sky was kind of creepy looking (and still is..). I decided it would be a great idea to use the fractal-clouds to texture the “fair-skies” with subtle clouds, and then switch to the bmp-clouds for story skies.

Since both Celestials and Fractal-Skies hack up the sky.cc and sky.h files considerably they provide their own to save one the headache. Unfortunately they were both written for TGE-1.2 not 1.3. This made for an interesting three-way merge that after long hours, I actually accomplished.

I eventually got it down to one variable: set it to 'true' and fractal-skies would be used when clouds are 'on' in-game, set it to 'false' and .bmp clouds would be used when clouds are on. This isn't the desired behavior, but a step in the right direction none-the-less... The rest has yet to be figured out.

III.6.2 Giving Back to the Community

In the end I was able to submit a resource but even that was not easily accomplished. Since I was developing on Windows, I didn't have CVS. So I had to copy all the files to Linux, import them, and produce a patch file. Also, do to time constraints, I didn't seriously test the resource before posting it. Some people have gotten it to work, other have had less luck...

III.6.3 Enviro-Torque 2.0

In my original Enviro-Torque Resource I refreshed the most recent Celestial day/night cycle and did a hack job of "integrating" it with a weather-resource to create more of a single *Dynamic Environment System.*

Now, I'm currently **designing** *Enviro-Torque 2.0*; a ground-up rewrite of my dynamic-environment system resource. It hopes to offer fully modular and scaleable management of

date/time, weather and celestial events[10]

The specific-feature wish list is fairly long but heading it are multiple celestial bodies, and multiple (independently configurable) weather zones. Also everything will be done from inside a single C++ class hierarchy, for better integration and faster performance.

Currently the class diagram stands at 14 classes. The basic run down is a scheduler on top which operates on a collection of task-objects. These include the Calendar-Module, the Weather Manager, Celestial Manager, and any user defined tasks large or small. Similarly the Weather Manager operates on a collection of weather-zone objects, while the Celestial-manager operates on a collection of celestial objects.

I intend to have several celestial-objects like *celSun.cc, celSatelite.cc, celStarfield.cc* all derived from *celestialObject.cc*. For celSun.cc, I really want to capture the essence of Sun.cc and fxSunLight.cc[11] into one class so everything that goes together IS together, and I can have multiple instance of it. I don't yet know if this is possible, depending on how tightly the rest of TGE is coupled to Sun.cc

The last and most nebulous component of the whole thing will be the numerous hooks from the celestial subsystem into various parts of the engine to create the actual day/night effects.....

III.6.4 Water Specularity

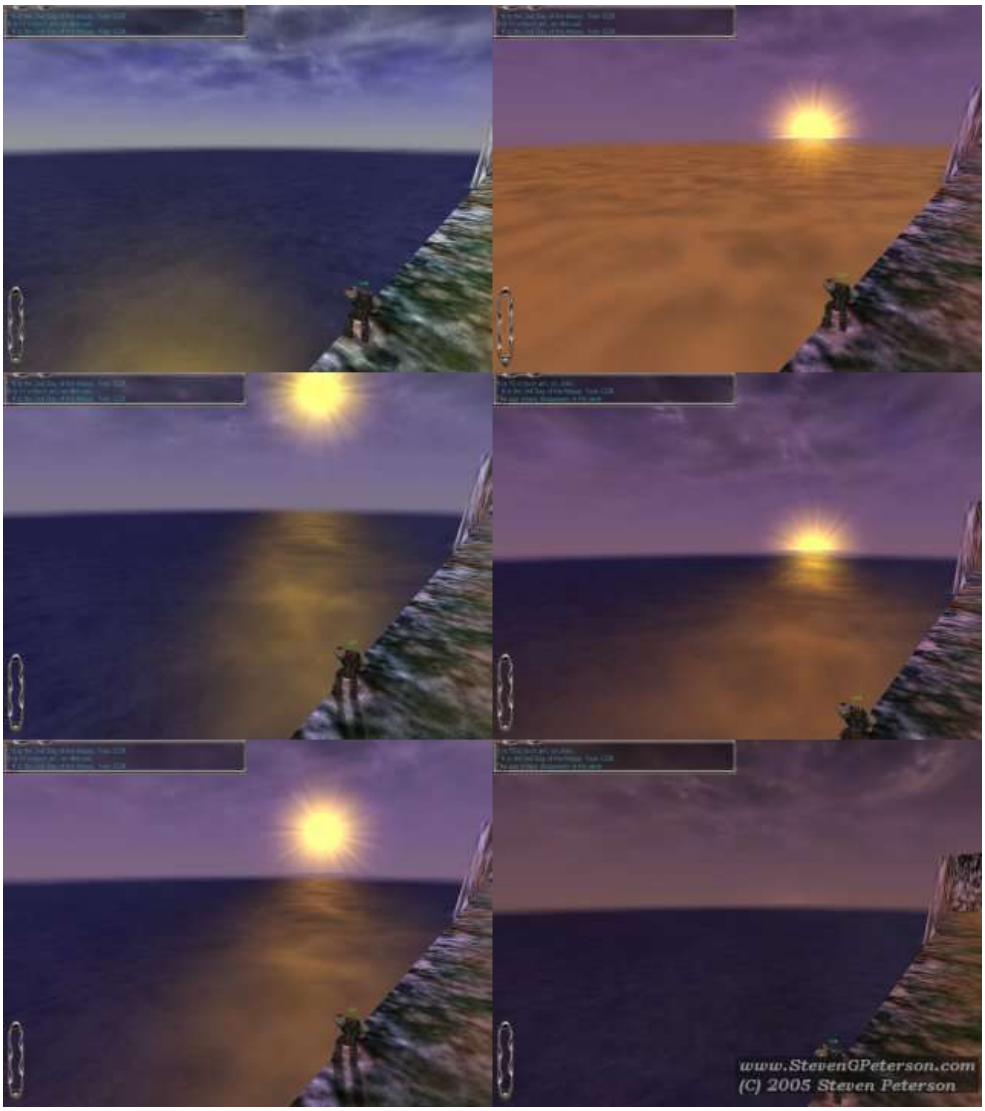
One of the many new features of much *smaller* scope is the sunset in fig. 2(below). This is an early prototype of a new hook from the celestial sub-system into the water-block to scale the specula highlighting as a function of the position of the sun.

In the final version I want to shade and scale the reflection using a weighted average of values from all celestial objects above the horizon, with those *closest* to the horizon getting the heaviest weight. This is just one of the many new features coming.

10 If I had defined this problem three months ago, it would have made a better, more concise and more programing centric senior-project. But alas, I did not...

11 Sun.cc is the engine class used to calculate basic terrain lighting and shadows based on the position of a sun. fxSunLight was originally a user-submitted resource, which has long since made it into the official CVS. It produces the actual sun-flare that you see in the sky.

(Fig. 3 specula Highlights of a Setting Sun)



III.7 Torque Lighting Kit

Shortly before submitting my first Enviro-Torque resource, I decided I wanted *more* out of my TGE. GarageGames provides the *Torque Lighting Kit* produced by a 3rd party developer (Synapse Games) as an expansion pack for TGE. It provides a whole bunch of new lighting effects and features, most of which I haven't gotten to explore.

Since the change list to the original engine code is so significant, a complete SDK is offered for download with the Lighting-Kit already merged into TGE 1.3. The new code base is dubbed TGE-1.3.5 and is completely worth the $50.

Next I had to merge all of my changes to the original engine base into the new engine base. This actually went very smoothly, it just took a little time. Working with Torque tends to make one really good at applying outdated CVS-patch files by hand...

Even before one actually reads the TLK documentation, the lighting kit starts enhancing one's 3D-world. Using over-exposure, it creates a highlighting effect on the terrain to brighten the world, and increase contrast between shadowed and non-shadowed areas. All in all the Torque-Lighting-Kit was a good buy on the road to building an immersive environment but it also added more variables to deal with on the way to a quick prototype.

III.8 Textures

The worst mistake I made was to say I wanted everything in the game to be mine from scratch. Even after cutting out the design phase, I failed to recognize how heavy the art work-load was in creating a game and that this project would have to be a quick prototype, not a complete game.

I tried to create every texture I needed on my own, which was quite the learning experience in itself. In the game I *did* create every texture, except for any wood/bark. The process was fairly intense and I followed several long tutorials, so it's somewhat beyond the scope of this paper but here's the overview.

There's two basic methods to create a texture. The first is from an up-close source-photo of some real-life texture. I took over a hundred of these with a digital camera, rocks, gravel, wood, water, brick, stone concrete etc.

In the GIMP the image has to be cropped and scaled to an appropriate size. Usually 256x256px or 512x512px. Getting the zoom right is a little bit of an art though, and particularly troublesome with terrain textures for Torque.

After that the first and last steps are usually making sure the image tiles correctly. The trick to this is to use the 'offset' tool and offset the image by (50% x, 50% y) so that u can see the seams. Then use the 'clone' tool to hide those seams.

By using *lots* of layers, different noise and blur filters, layer masks, and bump-mapping to give the image depth, some pretty cool effects can be achieved. Usually making one texture takes about an afternoon/evening and can produce numerous versions. This is how I created

textures like the rock for mountains, dirt, and and brick.

The other approach is to start with several layers of solid colors, and then add noise and filters to give them textures. Then apply noise to layer masks, to let lower layers show through the upper layers. This is the short version of how I created the grass texture.

In the end, textures are a long trial and error process. Once you have them, applying them wisely is another art unto itself, and not with out it's pit-falls. The first thing is to have textures that go together. Space-age and medieval textures should usually not be combine, nor should cell-shaded and photo-realistic in most cases, nor textures of entirely different resolutions.

Next thing to keep in mind is make sure the grain of textures is going the appropriate direction. For a wall this is usually vertical, for a floor that meets the wall, usually parallel to the wall, similar to the deck behind a house; or in a room, parallel to the longer wall. If the grains meet at odd angles it looks weird. It's also important to align the textures in a way that makes sense. For example, a brick wall should start with the first full brick on the floor, not half a brick.

To help hide alignment problems, and at any point where two different textures meet, there should be some kind of transition or border, just like the trim in a house, or the stripping between where tile and carpet meet. Other wise it looks bad.

I did this in several places in my level-one mock up. Notice on the outside, I used a stone texture, as a border between the green brick and the green ground. This actually solves several problems besides adding nice contrast. In this case the stone was a borrowed texture, but I will probably trade it for a brown version of the green brick in a future iteration. Inside the level I (consistently) used an oak trim all the way around the rooms. This provides that border/transition, and also hides seams where the brick doesn't line up nicely.

A moderate amount of variation in texture is also good to break things up. This isn't something I did well, but I probably could have found a different ceiling texture, to help keep things from blurring together so much. In addition, each labyrinth in a completed version of the game would have different textures and color schemes for variety.

III.9 Other Environment Modifications

III.9.1 Making a Tiled World Flat

The maximum size a Torque terrain can be equates to a 256 x 256px height map. However, Torque tiles that terrain, to give the player a seemingly infinite world. This is great unless the developer wants a finite world...

My 'Legend of Zelda' map has enough space north-to-south that one can't tell it's tiled in-game. East-to-west this is not the case. Several resources exist to fix this problem by simply not rendering terrain outside of the 'main block'. This fix is fairly complex but really comes down to one line of code where it decides not to render outside the main block. This leaves a great 'void' past the edge of the world. This toggle-able fix actually made it into the TGE-1.4 HEAD, but

isn't at all what I wanted.

Instead, I wanted an expansive ocean, a problem that no one seems to have been able to solve with Torque. I started going through and looking for everywhere in the code 'getHeight() is called. In each case I tried to check if I was looking at the 'main-block' or not which is done differently just about every time. If so, I forced height to be zero.

Following this strategy I made about a dozen changes to the rendering code, as opposed to the one change of the “sweep-it-under-the-rug” solution. In the end I was able to force repeated terrain to be toggled with zero elevation. However if you look at it, it's still somewhat buggy. I finally turned collision *off* but I wasn't able to force it to zero and I haven't been able to make the water repeat over the tiled terrain correctly.

This one's a work in progress, but I seem to be on the track of a more complete solution than anyone else has come up with.

III.9.2 fxFoliage – the Green-Thumb of Torque

Another one of my favorite features is that for creating foliage in Torque. Modeling high polygon foliage is time-consuming, and would hurt rendering, besides being annoying to collide with even if it were more realistic. Instead a trick known as *billboarding* is used. Basically a one polygon(2Dimensional) image is rendered, with no collision detection done on it so that the player can walk right through it. To keep it from looking completely terrible, the image is rotated to always face the player.

The main purpose for this is creating small plants, and sparse grass. It can also be used to create trees from a distance or other things. I used this to create the grass and wheat-field near the spawn point on the map. The advantage of this class, is that it while you specify and area and a density for the plant to appear in, it actually scales the number of plants to keep performance high and only renders what's currently visible.

Actually using the fxFoilage class isn't hard in principle. Basically I created a simple .jpg image of my plant, and put a black border around it. I then made a second .jpg image in black/white mode with the plant white. This image is used for *alpha-clipping* to determine which parts of the image should be displayed and which should be transparent. Put them in the right directory, instantiate the mission object and the code does the rest.

For a while I had one image working, and one not. I eventually realized that the image dimensions have to be powers of 2 (e.g. 128 x 256 will work, 128 x 196 will not). Using an illegal image size causes the whole image to show up as a white square in-game.

While this is easy to do, getting it to look good is not so easy from an artistic point of view. There's also a bug where in the bottom-pixel of the image is wrapped around to the top creating a 1 pixel. line above the plants. Leaving a transparent border all the way around the image was supposed to fix this but did not, so it's one more outstanding issue.

III.9.3 Character Properties

There are numerous member variables of the player class (approx. 50). Several are of interest. The first one I changed was the maximum slope a player could climb. The magic number ended up being 55 degrees. Other variables deal with the height a player can jump, how fast they can run, how hard they fall and the recovery time before they can move again once they hit the ground.

One of the effects I later tried to implement was a wall-running effect similar to the Prince of Persia. I figured if I could bind a function to a button press that changed the maximum slope to 90 for a maximum of 3000ms or so and also trigger an animation in which the player rotates 90 degrees around his y-axis and runs sideways, it would look like wall-running. Unfortunately, those player variables are rather well protected, and I couldn't figure out how to change them after initialization, let alone from where I needed to in script.

IV Building the User Interface

IV.1 Torque GUI Editor

Torque includes several in-game editors, written in Torque-Script, to help one out. Hitting the <F-10> key calls up the GUI-Editor, which is useful for making menu's, inventory screens, and HUD(Heads-Up-Display) controls.

The big trick with using the GUI editor is that elements have to be placed correctly or they will bounce around when the user changes screen-resolutions. Elements can be placed with absolute or relative positioning from any edge, or centered on either axis, however getting things to stick where one wants takes a little practice.

IV.2 The Main Menu

To build the Legend of Zelda splash-Screen and Menu was a several step process. I put a decent amount of time into this early on, when I still thought I had time to spare on things like marketing.

First I drew the image of the tri-force out on paper. It's a simple equilateral triangle with a smaller, inverted equilateral triangle inside which breaks upt the interior into four separate regions. I had several more complex images I was considering but simplicity won out. I'm not an artist, but with a ruler and a little 6^{th} grade geometry the triangles were no match for me.

It's interesting to note that even though Nintendo has classically displayed the three tri-forces as a 2D-pyramid, this doesn't make sense in 3-Dimensions, at least as far as I can tell. I guess this goes to show that while the laws of physics hold fast, even in video games, marketing on the other hand is above the laws of nature.

Having drawn my Triforce, re-traced it with a Sharpie and scanned it with a dying Lexmark it was time for the big-guns. In GIMP I retraced over all the lines to sharpen them up, and made liberal use of the eraser to remove any unwanted marks. I was going to use flat colors but stumbled upon a strata-effect that gave the image some depth. Added the text and it was back to the Torque-GUI-Editor.

I copied my background into the correct directory and set it as the new background image for *all* the main-menu dialogs for the game. The buttons had to be resized and placed in the center. Again this required a little finesse and a lot of testing to get it right in all resolutions.

The background music is added in Torque-Script. There is a good resource on Garage-Games describing the process, with improvements listed in the comments. The idea is to create an *audio-profile* which is essentially an object with a name, filename, and several other attributes. Then when the main-menu is loaded, call the load-sound function with that audio-profile. On game-load, kill all existing audio-profiles, before starting any new ones. The only difficult part is these three steps are implemented with three or four code changes in three different files. This is the kind of thing where good notes in the developer log would have been useful later on.

IV.3 Building an Inventory System

IV.3.1 Overview

The inventory was a particularly difficult concept to wrap my head around, for something I expected to be so easy. The (first)key was to realize there's two completely separate problems to be solved not one.

1) Creating an Inventory-GUI screen to interact with the inventory system.
2) Programming the functional inventory system.

The second key was to realize this may be a chicken-and-the-egg problem best solved by several iteration and not in one fell-swoop.

IV.3.2 GUI

The GUI is, as always, the interactive *front-end* of the inventory system. The first step is to create some mock-up art for the screen. The basics are a background image, I used a small black, semi-transparent square to be tiled. Also some container-frames an “empty-slot” image and basic images for the inventory items.

The next step is to layout the basic inventory screen. The first time around is just a prototype to show what's in the inventory for testing purposes. This is about as far as I ever got on this part of the program, but with the GUI-editor and some Torque-Script, it is possible to create rather complex and interactive Menus.

The last thing to remember is to go into the client/scripts/default.bind.cs file and bind a key to the new inventory GUI, usually “I”.

IV.3.3 Functionality

Torque includes a basic game shell to get one started called starter.fps which sets one well on the way to creating any first or third person game with proper network support. Included in it is a basic inventory implementation. Unfortunately there is *no* documentation on how this inventory system works. Should I ever actually figure it out I may post an article on the new GarageGames wikki but here's my take on it so far.

The inventory system is spread out across at least three or four inter-related files. While several of them offers verbose comments about what the file does, there's nothing to offer a “big-picture” of how their interrelated.

Basic inventory functions are defined in inventory.cs. Made more complex by Torque-Scripts loosely-scooped, typeless, define-as-you-go policies, it's not entirely clear where stuff is actually stored. It appears there is an aptly named 'inventory' object that acts as a collection of 'item' objects. At least it makes sense in concept.

The file item.cs defines functions that are specific to the 'item' class rather than

'inventory' class. [12] The file weapon.cs contains an examples of actual item definitions. There are at least two pre-defined item types that can be instantiated: 'ammo' and weapon'. This file also shows how sound effects and animations are attached to the object.

To keep things separate I created a new file treasure.cs which includes all of my more basic object instantiations. Having done this correctly, I was able to insert instances of each sprite-object into the world and walking over them automatically adds them to inventory. I never did get as far as selecting or using items.

IV.4 Adding the Orbit-Cam and 'Advanced Camera Controls'

Since I was really thinking of this game from a console perspective, I wanted to make it playable with a console style game-pad like the PS2-styled Logitech Dual-Action controller, which plugs in via USB.

The *starter.fps* shell-game is tuned to use to be used in a first-person-shooter format. This can easily be fixed though. I found an 'Advanced Camera' resource which implements more than half a dozen *different* third-person cameras. The one I was looking for was called *Orbit-Cam.*

The Orbit-Cam is set a fixed maximum distance from the center of the player model, this is the orbit-radius. If one imagines a sphere around the player with this radius, the camera can be rotated to any point on this sphere or orbit. The camera uses some basic collision detection and zooms in on the character-model appropriately, to avoid collision with walls or other objects.

Setting up the resource was fairly easy but left two problems. Without redefining the movement controls, moving the player left and right still caused the character to *strafe* left and right; the default movement for a first-person-view. Also all directional-controls were still relative to the character-model's point of view, not the camera.

Enter the 'Advanced Camera Controls' resource. This resource depends on the last one and is mainly targeted at fixing joystick controls. Essentially it takes a directional input from the joystick and *first* rotates the player to face that direction *relative to the camera's point of view* and *then* tell him to run forward in that direction. The process in which this is done is mildly complex and again requires some changes to the engine code, but when done, the joystick can be programmed real easily from script.

All of these changes leave us with a beautiful third-person action view and the potential for wonderfully intuitive and smooth controls. Now to program them.

IV.5 Programming the Game-Pad Joysticks

Having completed the last section, it's possible to program the game-pad as desired, entirely from Torque-Script. Programming on this high a level takes out some of the difficulty, but it's an inside look at what things are like where the rubber meets the road.

12It should be noted the concepts of objects and classes are only loosely defined in Torque-Script, so this is all appropriate at best.

The first thing I did was add a new function to defautl.bind.cs. I wanted to add additional bind.cs files to a list in the beginning of default.bind.cs but execute them last. Then i created gamepad.bind.cs to keep everything organized.

Buttons are relatively easy to program, just mapping them to a specific function. I set L1 to toggle to First-Person mode, and the 'A' button to jump. The joysticks are a little more complex. The joysticks are analog and return values between 0 and 255 on each axis. Each axis of each joystick must be programmed separately.

All joystick input-values are tested against a variable called 'deadZone'. If the absolute value of the input is less than 'deadZone' we throw it out. This creates a dead spot at the joysticks center, otherwise it would seem way over-sensitive and never sit still. Next the joystick values are multiplied by a 'sensitivity' value that becomes a part of player-preferences.

The left joystick is used for player movement, which is pretty basic to program since the 'Advanced Camera Movement' resource will do the hard work for us in the engine.

The Right joystick has several extra complication. First it tests to see if it's in first or third person mode. Second, in either mode, which way the camera should move corresponding to a positive or negative y-axis value is subject to interpretation. This also true for x-axis values in third-person mode. Therefore it checks a boolean player-preference value to see if there should be a spin-invert or look invert. When everything is done it calls the move function and feeds it the correct value.

IV.6 Final Design

The final controls didn't quite live up to the aspirations of the design document. The biggest problem was that I couldn't program new moves using the included character model. A lot of the programming for player movement, combat, and item-use, were it to do anything at all visually, would require a new character model with new animations. As I detail later, this proved to be more than I could manage.

(Table 1 – Control Scheme – Final Concept)

Move	L-Joystick
Camera Control	R-Joystick
FPS-mode Z-Targeting (-)	L2
FPS-mode Z-Targeting (+)	R2
Primary Attack	[X]
Secondary Attack	[Square]
Jump/Grab/Pull-up	[Triangle]
Roll/Drop/Crouch	[Circle]
FPS-mode Toggle (hold-down)	L1
3rd-Person Mode: Wall-Run	R1
1st-Person Mode: Secondary Attack	R1
Function Key (if needed)	R2
Quick Weapon Select	L2

V 3rd- Party Tools

V.1 Blender

V.1.1 Blender Overview

Blender is a character modeling and animation studio along the lines of 3DS-Max. Unlike it's competition however, Blender is available freely at www.Blender.org. It was originally a commercial project whose company went under, but not before Blender found a devoted fan base. In the early 2000's a deal was struck with the investors who now owned Blender to release it open-source for $100,000. The money was raised through donations in seven short weeks and development on it continues to this day.

Anyone who uses complex software realizes there's a direct relationship between how powerful the software is and how steep the learning curve is. Blender is an extremely powerful and versatile application. In fact the analogy that came to me is the VI. Text-editor. While VI. gives the programmer absolute control over 1-dimension, Blender gives the same level of detailed control, multiplied by three dimensions.

To make it more interesting, the Blender GUI is entirely custom and non-standard. It is well thought out, and those who have learned it swear by it, insisting that it makes for a fast and fluid work-flow. Unfortunately for the new-comer, it is completely unintuitive. Luckily there are a good number of online documents and tutorials, and the book (which I bought) is pretty good.

Largely due to the learning curve, and the long process of creating and exporting a model, Blender is one of the things I dumped the most time into, even though nothing from it has yet made it into my final project.

Blender has several extra features that really push it to the next level. It includes it's own still-life render-engine which seems to work really well. Supposedly there is a basic built in game-engine that is in the works or partially implemented, but hasn't seen the light of day yet. Blender also has complete support for plugin-modules and Perl-scripts which make it highly customizable and extendable.

V.1.2 Building the Raft

Originally I was going to build 'Link', all enemies, NPC models, and also all items/weapons in Blender. The one item model that did get built was the raft. It was constructed very simply from ½ a dozen cylinders made to look like logs. Then I took 2 short cylinders and stretched them to encompass all six logs to look like wide ribbons holding the thing together.

The next step was texturing it. I already had a bark texture I had made, but getting it to align properly took awhile. Blender has a wide range of tools for creating materials and textures, and adjusting how light reacts to them. But again, finding the right way to use the right tool is something of a trial and error process.

Blender's render-engine let me create rendered-images of my raft to post online.

Exporting objects to Torque, however, isn't so easy. There is a special Perl-Script that lets one export to the correct file format. However a lot of the Blender-object's properties have to be configured in a specific fashion to allow it to be exported. After getting the Perl-script installed, no easy task in itself, I was able to export the included example file. I was never able to export my raft though. In the end, I didn't have the time to invest in learning how the export module actually worked.

V.1.3 MakeHuman

Make human is a software package that takes advantage of Blender's incredible extensibility. First it provides you with a complete model of a generic human-being. It then adds a new GUI-layer that let you edit the model with point and click ease.

The idea is they've defined numerous *targets* on the model. Targets can be adjusted in a specific fashion, using a sliding scale. Targets may make muscles bigger or smaller, parts of the body taller, shorter, fatter, skinnier, wider, narrower, or move features like cheek bones and eyebrows in or out and up or down etc.

The original Blender demo that was released in the spring only supplied the targets for the face. Targets are apparently defined by extra files in a special dir. tree in the MakeHuman folder. Later a stand-alone demo was released that had targets for almost the entire body, but this was not integrated with Blender. What I did was take all the *target* files and a few other and merged them into the tree for my *Blender MakeHuman* and it worked! I was now able to edit the entire thing in Blender.

This is great for someone like me who isn't a modeling pro and not a real artist. Supposedly the MakeHuman model is also 'fully-rigged' meaning it has the pseudo-skeleton a model needs to be animated. Just another head-start to save me time!

V.1.4 Creating Link

Armed with the knowledge from the tutorials and the power of *MakeHuman Blender*, I set out to create 'Link', my main character. Alas the learning curves had been too steep, the setup to involved and I ran out of time before getting to cloth, animate or texture my model. This was a huge setback because without the animated model, I couldn't do things like program new player moves. Since I was altogether out of time at this point, it didn't really matter though.

V.1.5 Notes on Trees

Trees are a rather complex problem to solve in a video game. The first issue is how does one create a tree that looks realistic? And can this be reduced to an algorithm to randomly generate plants/trees?

The second problem is polygon-count vs. realism. Using *Billboarding* (mentioned above) one can create a really great 1 or 2 polygon tree, that will look wonderful from a distance. It's also possible to create a realistic 20,000 polygon tree that will look good from any angle and will support full collision. Should the player jump onto the tree from above, he will hit every branch

on the way down. A whole forest of these though would be virtually impossible to render.

The trees I ended up using are the default ones included with Torque. I hate them. The approach is to model the tree-trunk and major branches. Then uses between four and eight large polygons, textured with the image of hundreds of leaves, offset from each other to create the top of the tree. To the method's credit it has been used in such best-selling titles as Grand-Theft-Auto amongst a hundred others. This doesn't change the fact that it's the stupidest thing I've ever seen.

Awhile back I was playing with the *Povray* Vector-Rendering system, which is used for creating 3D models for still-life renders or basic animations. At the time I ran into a tree generating program I liked.[13] While the unnamed program seems to have fallen out of development, the source-code is still online, and he has a PDF of his source-paper on the topic.

There is another system called the "L-System" which seems to be popular for tree generation. I believe it's based on the generation of mathematical trees and can be used for numerous things, but I don't know much about it yet.

I was hoping to do some research on these methods and write a basic python program for Blender to do tree generation. Once again this fell victim to the axe of time.[14] Until then I'll have to suffer with the horribly unrealistic trees.

V.2 QuArK

V.2.1 QuArK Overview

QuArK stands for *Quake Army Knife* and is available from http://dynamic.gamespy.com/~quark/. QuArK is a modeling program for building interiors and large objects. As opposed to character models or weapons that require high polygon-counts and animation but exist inside a simple collision-box; interiors are meant to be low polygon-count, and static, but have detailed collision. This makes sense when you think about it long enough.

Most of the buildings in video-games are rather simple, and usually simpler than they look. In most cases, all walls, floors and ceilings are but a single *brush*, usually a misshapen six-sided cube. All the detail and depth is provided by the artist-texture applied to each face after the fact.

While far quicker to pick up than Blender, this kind of modeling is still time-consuming. QuArK now supports a wide range of engines and gets the job done, for free, but it's generally an annoying program to use. GarageGames is supposed to release a new program to do this exact job – better, but until then...

13 http://www.infa.abo.fi/~jweeriks/treegen/
14 "Axe of Time"... hmm, sounds like a candidate for one of X-Play's EB-sponsored "weird games of the week".

V.2.2 A Test Room

When Quark starts, it presents a test room to get one started. It was approximately the correct scale, and some trial-and error helped get things right. Once again, the export module for Torque had to be installed and configured, it's a program called *map2diff*. This time, however the overall setup and use was much easier than for Blender.

I expanded the basic room and applied a texture to the walls, ceiling and floor. I then added trim all the way around the room, just to break it up a little. I also framed out a door that I could copy and paste to any wall. This was my basic room.

Once I had a room-template and a door-template, adding more rooms to my level was simple. This isn't to say simple and fast are equivalent though the process did get smoother with practice. I eventually mapped out half of level one. Not a complete level, but enough for proof of concept.

Unfortunately, the final time I imported to Torque, the level looked *"*full bright*"* which suggests there's a 'hole' in the building between two walls somewhere. Any leak can cause all sorts of problems, and even doors have to be sealed off with 'portals' or transparent brushes that represent a transition but seal off a leak. In this case the problem should be easily fixable but isn't near the top of the to-do list.

V.2.3 Into the Furor Bunker

I needed to create some kind of entrance into the level from above ground. This presented numerous problems, particularly for level one, whose entrance is on an island. The idea was to get the player (and the tunnel he was in) below ground level in as short a horizontal distance as possible.

The solution came in the form of a History-Channel documentary on the *Furor Bunker*. This was Hitler's famous last-ditch command bunker where he spent the last six months of the war; hundreds of feet below ground and protected by 50' thick concrete walls. Though the allies knew of the bunker's existence, it wasn't until Berlin was actually liberated that it was found beneath the Chancellory Building.

The images of the dark, narrow, steep, stairs going down into the bunker seemed perfect for my problem. Though the sense of scale isn't really the same that's where the idea came from.

V.2.4 Solving Problems with Torque-Triggers

V.2.4.1 About Triggers

A trigger in Torque is essentially a defined 3D-box that can trigger a functional event when the player enters it. Their useful for solving all sorts of interesting problems. Even though Torque allows for seamless transitions between outdoors and indoors, I had several such problems to solve when the player entered a labyrinth.

V.2.4.2 Teleports

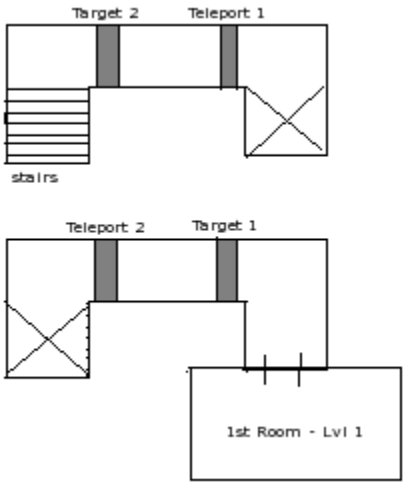
The first problem I had was that the geometry of the ground didn't always match the geometry of the level. This is particularly notable in level one where the entrance is on an island.

The solution to this was to create an horseshoe-shaped entrance that led to a dead-end rather than an actual building. Separately, I created the real level one, with the same horseshoe shaped hallway leading to another dead-end rather than the stairs going out. The first structure I placed under the island, while the latter I tucked neatly behind Mt. Doom where the player will never see it. It actually hovers unrealistically above the water, as walls don't stop water in TGE.

The next step uses four triggers: two teleporters and two targets. A trigger can be made to do anything so I used a bunch of them. One of the tricks was if I just used one pair of two-way-teleporters, if the player walked into the teleport(and teleported) and then walked backwards out of the trigger, he would now be on the wrong side, no-good.

The figure below shows the correct layout. When the player enters the level (top image), he hits 'Teleport 1' and is teleported to 'Target 1' in the real labyrinth building. He can then round the corner without knowing he changed buildings. Then going back the other way the player is fine till he hits 'Teleport 2' at which point he's seamlessly bounced back to 'Target 2' and can exit the building again.

(fig. 4, Teleport setup for level entrance)



This is something I got working really well particularly in first person mode. However, a bug in the 3rd-person orbit-cam ruins the effect from that view. The camera is supposed to "follow" the player around the map by always trying to play catch-up to the player. Unfortunately there's no error checking to snap the camera to the correct position if the distance gets to be too great. On teleport, this causes the orbit cam to fly across Hyrule, through Mt. Doom, over the ocean, through the walls of labyrinth one and up to the player who was instantly teleported. I took several passes at trying to fix this but never managed to get it right.

V.2.4.3 Spawn-Point Selection

The second use I had for triggers never got implemented once i figured it out conceptually, but it would be trivial to add. The idea is that when the player dies anywhere in the *Overworld* he re-spawns back at the starting spawn point. However, if he dies in a labyrinth, then he should re-spawn in the first room of the labyrinth.

This can be accomplished with two triggers at the level entrance. When the player enters the outer-most trigger a 'spawn-variable' is set to 0 representing the overworld. When the player enters the inner-most trigger, the variable is set to the level number. Then when the player dies, a switch-statement could easily determine which spawn-point should be used.

VI Documentation

VI.1 Documentation List

All project documentation is made available at:

- http://stevengpeterson.com/zelda (project-page)
- http://stevengpeterson.com/?seniorProject&zelda-documentation

In the end there will be:

- Project Proposal
- To-Do List
- Design-Doc
- Developers-Log
- Final Paper (this document)

VI.2 Design Doc Issues

The design document proved to be a difficult to maintain living-document. While a lot of it came straight from the original game manual the interesting details I mostly made up on the fly. Eventually the official design-doc fell by the way side, the real work was happening day-by-day on the yellow-pads. As features either got implemented, or fell off the cut-list for this iteration, the final drafts were safely merged into the official design-doc.

VI.3 Web-Site

After a few problems I wrote a new web-site to showcase all of my work, current projects, portfolio and resume. You can visit me online at:

- http://stevengpeterson.com
- http://stevengpeterson.net

VI.4 Resume

As per the original goal the resume got a face-lift. It's now available right here:

- http://stevengpeterson.com/?aboutme&resume

VI.5 Portfolio

Projects are still being added to the portfolio, but it is available at:

- http://stevengpeterson.com/?portfolio&portfolio