The Shaman Engines Postmortem

By
Tommy A. Brosman IV
John Stone
Thom Strandberg
Alex Chen

Editor
Tommy A. Brosman IV

May 10, 2010

Table of Contents

Hammertank Games

The Vision

"The Shaman Engines" was first conceived in January 2009. We started with a team of four, which included the Technical Director (Tommy) and the Designer (Thom). Tommy had recently worked with members of the team on previous projects ("Bloxel Blast" among them). The core technology was built mainly from an engine he actively maintained and used, originally created for a graphics class taken during the Fall 2008 Semester. This in turn was adapted from an even older set of code from "Vectron Assault," a project dating back to September 2007. In short, there was a good amount of proven, tested base technology and patterns.

Everyone had concepts for a design, with initially very few commonalities. There were ideas for a summoning system, a mech shooter, and a racing game being thrown around. After combining a few different concepts, the team came up with a shared vision. Thus, Hammertank Games was born. Every Friday at noon (a tradition that continued until the end of the development), the team would meet, brainstorm, assign tasks, and plan out what we would do next. We began to draft a story and a technical overview. The overview isolated key responsibilities into modules, forming the basis for the TDD. The story helped to shape the design and provided key information about the level design and art style

Early on in the summer, half the team split off. The two groups maintained their relationship, giving each other technical advice and suggestions for improvement on their projects. Until fall, very little was done, with the rationale that certain elements of the design would change radically once we gained new members. Thom and Tommy spent a few Friday and weekends near the end of the summer working on the technical design elements while communicating via Skype.

Early Development

In September 2009, Hammertank gained 3 new members, including Alex Chen and John Stone. Initially, Tommy took the role of Physics Programer/Technical Director, with Alex and John as general programmers. At the beginning of the main development cycle, the modules and code elements were highly coupled (everyone needed things from the event system, the game logic depended on the input system, etc). To deal with this, a Gantt chart was drafted and maintained for the first 5 or 6 weeks of development. This was instrumental in allocating resources and finishing the basic engine in a timely fashion. The Technical Design Document was finished within the first few weeks of development, providing a solid reference for the engine's core structure, the coding standards, and the specific library versions we were using. This document was actively maintained throughout the first semester, avoiding many of the pitfalls commonly associated with iterative game development.

When it became clear we would not find artists anytime soon, Thom began work on an animated mech (the "Heavy Mech"). John worked on the FBX importer, dealing with the oddities of the API in order to import animated 3D models and serialize them to our binary format (see "What Went Right" under "Technical Overview").

Engine Proof rolled around in early November. The engine had animation features, basic physics, input handling, an early version of the GUI, and asset loading. It was also very rough, with many components unfinished, specifically the game's internal state machine. Midway through November, the team was reorganized and paired down to four members. John Stone became the producer, and Tommy became the graphics programmer (as well as the physics programmer). This strained our time budget, and by the time the First Playable milestone came around, very few visible additions had been made (non-flat terrain, enemy spawning, some basic AI). It was clear at that point

that we would have to work over winter break.

Wintersession

During the last three weeks of winter break, we attacked the tools (see “Bottlenecks in the level editor development” under “What Went Wrong” in the “Technical Overview”). This made for a lot of long days/nights and missed sleep, in spite of the fact we were on vacation. During this time, Tommy tuned and tweaked the model buffers, studying the OpenGL API specification and extensions to figure out how to best maximize compatibility and speed. John, Alex, and Tommy all tackled bugs in the level editor, while Thom worked on improving elements of the GDD (specifically the interface layout and level flow overview). It was also during this time that Tommy developed the basis for the particle system. This was later expanded on by Thom to create the explosions that can be seen in the final game.

Making The Game

After winter break, we continued on towards Alpha. The graphics pipeline was slowly being cleaned up (and in some places, re-written), a process that continued until Beta. The physics were mostly finalized, with a few issues related to terrain following. These were avoided by changing the design (not the engine). Thom finished off a second mech (the light mech), Alex iterated on the GUI, and John continued working on the level editor (into which we eventually integrated the model importer).

Near mid-January, we added four artists to our team. Among these was Roberto Rodriguez, who became our environmental artist for the city level. Thom improved on the old hard-coded methods of instantiating enemies, creating a flexible object creation system with factory patterns loaded from XML. By the time we made Alpha, we had mocked up level 3 (a feat accomplished a few nights before the presentation using an ascii-based map parser that generated XML). We had created a city full of enemies, and the game was no longer just an engine. We were finally seeing the results of our hard work, and they were paying off. We play-tested heavily between Alpha and Beta, as well as beginning the other two levels.

By Beta, we had our 3 levels. The main complaints were aesthetic, specifically the sound and the GUI. Tommy addressed the sound issues by mixing together a set of effects from the sound libraries DigiPen owned. Alex made the GUI pretty while tackling issues related to resolution-changing. Roberto had developed a set of assets for level 3, and continued to iterate on the textures until a few weeks before Gold. Thom worked on the AI, adding a nav graph, then improved the enemies themselves, varying their weapons. In the last few weeks of development, a new game mode was added (“infinite mode” at the end of level 3), loading/etc was tweaked to pass TCRs, the AI was tweaked some more, fixes were made, and we all went into crunch mode.

By Gold, only one of our four artists delivered working assets (Roberto), we had no major upgrades on either level 1 or 2, and we had to turn in another build the day after due to a resolution changing bug (that only occurred in Release). Despite these setbacks, we had created a solid game that was fun to play and embodied our high concept.

Production Overview

What Went Right

The Dev Team – We had a team that had a great diversity of different skills to bring to the table. Everyone had different ideas about how to do different things and all of our ideas came together to become The Shaman Engines. Tommy Brosman, our Tech. Director, had a fair amount of previous experience. Typically when things broke, he would either come up with a fix or a workaround. John Stone, our Producer, kept very good track of the progress of our game. Sometimes when things needed to be in the game, even when they weren't scheduled, he noticed them and wrote them into the game. Alex Chen, our GUI Programmer, always got his tasks done. Thom Strandberg, our Game Designer, had a lot of creative ideas that went into making the game as great as it is.

Roberto Rodriguez – Roberto constantly was outdoing himself. From the time, we first saw his concept art til the end where he gave us his final building versions, we have been extremely pleased with the quality of his work.

Agile Iterative Periods – We had a set idea of what we wanted to accomplish in our game. This allowed us to keep our focus centered around what we wanted, which was a fast-paced mech FPS. We decided that the first thing we needed to give our game a good pace, was to have excellent controls and a solid engine. The entire first semester we spent building up our engine. At first playable we were still building our foundation, so we might have been a little behind everyone else at the time, however having such a solid engine allowed us to primarily focus on design implementation the second semester.

We had playtests for several hours nearly every week. The playtests gave us great feedback and let us and make changes almost instantaneously to reflect the ideas that the players gave us. We also made sure that there wasn't a week that passed that we hadn't bettered our game. We constantly built upon what we did the week before and always had a great progression. We kept our planning flexible enough to drop design ideas, assets, and other things we would have liked to have gotten into our game, but didn't have enough time to implement.

Avoiding Overcomplicated Task Systems – Our tasks were written in an Excel spreadsheet from Day 1. Early in the engine development, when we were blocking out core systems, we had a Gantt chart. We continued to use this for the first 5 weeks of development until the dependencies were largely non-existent. In this case, a Gantt chart made sense: when you begin, you have dependencies everywhere. But, if you're writing your engine in a sane manner, those dependencies should disappear. One of the key benefits of modular architecture is modular development.

For all tasks, even those we used the Gantt chart to better understand, there was an entry in our spreadsheet. The goes like this: task #, description of task, assigned person(s), started date, finished date, expected finish date, notes. If a task is overdue, the taskmaster (in this case, Tommy) changes the text to a red color. Late tasks are not changed back to black when they are finished. Instead, they are used to better understand the team's workflow. There are 3 main sections with tasks in them. Current (unfinished) tasks, recently finished tasks, and old (finished) tasks.

The development of “The Shaman Engines” consisted of 270 discrete tasks. Some were skipped, always with a note giving the date (and possibly the reason). Some were done by other people, again this was recorded using the Notes section. In essence, the entire system is an augmented checklist (or a linear Gantt-chart, if you are so inclined).

What Went Wrong

Difficulty with Communications – Some members on our team had issues with answering their phone and checking their e-mails. The biggest problem with this is that sometimes it would be several days before we could re-establish communications with them. This blocked us severely at critical times when we needed to know how to use a portion of their code. Some of the members on our team would not show up regularly to our team meetings, and one stopped going altogether.

Artist Moratorium – We as a mitigation for the risk of adding artists to our team so late in the project made our plans on a very skeptical note. We planned as though the artists were not going to get us anything. Even though we had planned this way, we were still disappointed for the most part that our artists didn't get us anything until the Friday before Gold. We had to drop 2 models because we didn't have enough time to tweak the files with all the other work we still had to do for our game.

If we had gotten our art assets only one week sooner, we possibly could have had time to work it into our game. The week before we ship is just far too late. We think that because of the fact that the game project was only 20% of the grade the artists also felt less inclined to meet the easy deadlines we gave them, moratorium aside. We think in the future we'd like to be able to possibly join the game course and the CG-300/350 course. Most of the artists who go to DigiPen we feel want to work on games, but feel as though creating excellent art for their game course is as important as the rest of their work for their portfolios.

That's not the feeling that should come across from a co-joined team. We've heard from other artists in other teams mentioning something along the same lines. To bring us both together was a great idea and we're grateful that it we have had artists to work with us. That being said, to truly have a unified school with less distinction between the artists and programmers, we need to be able to work together more, as well as know our boundaries with regards to how much control we all have over our teams.

There is one artist on our team that hasn't submitted anything for us. He continuously missed our team meetings. With the lack of understanding our boundaries, we as programmers were wondering what the steps would be for firing artists who fail to meet deadlines. Are we allowed to? We know for programmers, if it's not working out we can let them go as we have done last semester. We're not sure if we're able to for our artists as well.

Not enough planning on UI, Game States – Interface objects were not planned out adequately ahead of time, which resulted in at least one major overhaul of the GUI system, both in terms of design and code.

Game states were also poorly planned. The interface manager handled its own states separately from the normal game state, which led to some problems as the game developed. When changing an interface state, the interface manager would tell the game state its current state. The game state would switch interface states, causing more than a few debugging headaches.

Technical Overview

What Went Right

TDD and Documenting Before Developing – Writing a game without an overview of the architecture is building a house without a blueprint. Some people might try to throw the roof on first, other people will attempt to build walls in odd places, and in the end the best case for your pain and suffering is something future generations will term "Modern Art." Not so with our game. Although most of it began development in September 2009, the engine was planned out from January through August 2009. "Why 8 months?" "Why all the planning?" And of course, "That's not Agile!" The fact of the matter is, a lack of foresight usually means a lack of product. In the case of our game, it was even more vital. Half of our team quit during the design phase because they didn't want to work on the project. If a change like that had occurred in the thick of development, it would have surely been the end of the project as we knew it.

As student games go, the scale of our engine is staggering. It animates, does physics, graphics, collisions, AI... and most things without any sort of 3rd party library. It was also built with threading in mind. We use blocking model, and heavily synchronize only the object manager itself. This was planned out far in advance, with diagrams showing communications between various modules in the system. We had, from the very beginning, created a set of concepts to describe how things operated. Game Modules were mostly asynchronous, with a largely private interface. Core Modules were synchronized, and could be accessed by Game Modules or other Core Modules.

We also came up with a list of things we needed in terms of 3rd party libraries and tools. This evolved over time to match our needs, and in some cases, compensate for broken software. Everyone in the team would be able to give their opinion about a tool before we added it to our list. This became increasingly important as time went on, and it became apparent that we should limit the number of tools we were going to develop ourselves. Originally a physics testing tool, a mech construction tool, and an animating tool were on our list of things to build. These are all non-trivial tools, and would have required a lot of time if we had gone with them. In the latter half of the development phase (December 2009 – April 2010), our team lost another member, shrinking to only 4 developers. Knowing what we needed and what to change was important.

Some people argue against documenting early because "the game changes too much." It is for this very reason you should document early. How else will you re-factor your core architecture and toolchain?

The Level Editor – Without the level editor, we probably would have only been able to create one level. Though the level editor nearly isn't as verbose as we would have liked, it fulfilled all the requirements we had of it. Not only were we able to place, remove, and pick objects, but we were able to create ground meshes, convert, and extract FBX scenes. At the beginning of the second half of our development phase, we merged the level editor and the game's source base, so that only a few files were unique to the level editor. This allowed for changes to the main source base to propagate easily through the toolchain.

Bones vs. Hierarchical Animation – Originally we wanted a hierarchical animation system with solids (it is a mech shooter, after all). After a few conversations with Chris Peters, it became apparent that this was a bad idea. In terms of efficiency, bones are simply the way to go. More importantly, animating these hierarchical models would have required much trickery on the part of our mech artist (Thom Strandberg) and/or on the part of our toolchain. The straightforward approach is usually the best approach.

Platform-Neutral Code – Our game is platform-neutral. Everything that touches the OS/hardware does so through a platform-neutral interface (SDL, OpenGL, FMOD, etc). As a result, our code is understandable and pretty. Although this development practice was pre-texted as being necessary for cross-platform compatibility, the core reasoning is that overly platform-cohesive code is bad. “Vectron Assault” was written using lots of WGL and GDI code underneath the actual game. Hacking in proper resolution changing, etc, was absolutely hellish. System-specific interfaces are never pretty, and all of them are different. Unless you're developing for a console, or really really want to use DirectX, avoid overcomplicating your development cycle and go cross-platform.

Libraries and Tools – FBX is huge. Why do you need 60 MB libraries to parse a file? Despite its lack of built-in support and the fact that it is XML-based, COLLADA almost seems like a better option. On the other hand, when done right, FBX handles both 3DS MAX and Maya files without too much issue. Our toolchain can import models, skeletons, and animations from both without too many issues. Overall, it's a net win.

Boost is also huge. From the beginning, the Technical Director (Tommy) enforced the learning and use of Boost. The simple reason for this is C++ is incomplete (pre-C++0x, at least). There are no hash tables and there is no native threading support, already a huge strike against it. People who build games typically spend a lot of time making their own containers/etc. This can have disastrous consequences, especially if their unit tests do not have adequate code coverage. The tradeoff works like this: allow for awful compilation times (Boost, despite what the manual says, cannot be disassembled into its components and used in a modular fashion) and speed up development. In retrospect, some containers should have been avoided (multimap seems suspicious in its operation, especially when in the glaring light of the profiler) and possibly re-written for the purposes of the game.

SWIG is absolutely amazing once you get it to work. Getting it to work isn't terribly hard once you know the syntax. It manages garbage collection, auto-generates glue functions, handles pointer/reference conversion, and in general makes Lua bindings much easier to manage (especially with changing code). The .i files it requires are basically slightly modified versions of your header files. Once you have those, you just compile a C file and throw it in your project. There's no template magic or Lua bytecode trickery, just good old-fashioned ANSI C.

SDL is quite comprehensive in spite of its flaws (see the section on GLFW in “What Went Wrong”). You can change resolutions, deal with input, query information about the video hardware, and do so in a platform-neutral way. It is well-established, so there is a good amount of documentation/etc on it, and it runs on everything. It's a bit arcane in parts, but if you're using it as a hardware layer (and not trying to build the entire graphics engine in it), it works as intended.

What Went Wrong

Libraries and Tools – wxWidgets ended up taking far too much of our time and introduced quite a few bugs when we used it in coordination with SDL. After looking back, we realized we could have saved a lot of time by using AntTweakBar.

TiCpp was another library that took way too much time. In the case of TiCpp, it became evident near the end that the Windows version of the library was terribly, horribly broken. After wrestling with it, going through the source, cursing the developers who maintained their *NIX port so nicely but hated Windows, we gave up and went back to TinyXML. TinyXML works nicely, though it is not a DOM parser and has its fair share of issues. After a bit of research, Rapid XML seems like it might be a better option.

GLFW is a wonderful, lightweight library. That said, you cannot change the resolution while it is running. This is a huge design flaw, one that prevented us from using it for windowing. We do, however, continue to use it for its high-precision timer, which SDL apparently lacks on Windows (it has one, but it doesn't work properly as of SDL 1.3).

Bottlenecks in the level editor development – Alex, the GUI developer, was originally the sole resource assigned to the level editor. After a good deal of research and talking to other teams using widgets, he realized there weren't many resources on how to use widgets for games, at least not in the way we wanted to use it (see *What Went Wrong: Libraries and Tools*). Our goal was to use the existing graphics code to render the game in the editor window. The first hurdle was hijacking the widgets event loop. wxWidgets runs on its own event loop by default, which does not terminate until the application closes. Because it stole the main thread, it would keep the game from running. About 3 months into the main development cycle (which consisted of the last 8 months of the project), he fixed the problem by overriding the loop function in widgets and processing messages in a peek-handle-continue fashion.

This was not the end of our problems with wxWidgets. It became apparent that we needed a good way to render the game in a widgets panel. When focus was shifted between a separate widgets window and the game window, you could not regain it. During the last half of our winter break, the technical director called everyone up and told them to have the level editor fixed by the start of school, or drop it entirely. Early January saw the maturation of the level editor project, with the team crunching to fix it. There was a glCanvas object, but for some reason, it wouldn't behave as intended. It was then that we discovered that it was excluded from the build of wxWidgets we were using. The WIN32 define around the code was incorrect, and the flag to include glCanvas support was set to 0. After fixing this (an issue that was frequently re-fixed for all the boxes we worked on when developing, see *Roaming Profiles, DigiPen's SVN, and The Network*) we pushed ahead, setting up the glCanvas for rendering. It worked, it was functional, and somewhat useable (it became more usable when OBBs were used for picking, and the picking bugs were fixed).

Event System – Event systems are good things. Tuple spaces are good concepts. However, intercepting events as passive messages is probably not a terribly great idea unless there's some sort of binding code. You have to check names on both ends and everything is done using strings. The biggest problem with our event system was a bit more complex than this. For our event system, boost::any was used to store parameters. This sounds like a good idea until you look at the speed. Storing message parameters in a polymorphic fashion requires vtable lookups. Also, the excessive use of strings in the event system put further strain on the CPU. Chunks of raw binary data or delegates would probably yield better results in terms of both performance and useability.

Putting everything in one project – This was a huge mistake. Our game is around 30,000 lines of code (not include scripts, SWIG-generated lua wrappers, shaders, etc). We also employ a large

amount of code-level optimization to get the game to run at a decent speed (in-lining functions, stripping out frame information, etc). On a decent machine (at the time of this writing, 2 Ghz Core 2 Duo) the compilation time is 5 minutes. Considering the way DigiPen uses roaming profiles for everything (with a profile size limit of 30 MB), rebuilding every time usually meant a short lunch break. Near the end, many modules remained the same without much change. There was no way with the project structure we were using to store these as libraries on the server. This is something everyone knew was a bad idea, but never set aside time to change.

Roaming Profiles, DigiPen's SVN, and The Network – This project is a good example of why DigiPen will be requiring laptops soon. Checking everything out every time on SVN is a terrible idea, but for those of us who didn't bring laptops (roughly half), this was exactly what happened. The roaming profiles hold up to 30 MB (most of which is filled by application-specific garbage), our SVN repository is well over 200 MB. It was structured in such a way that you only need to check out the project directory unless you're updating tasks/notes/etc, but that still requires around 100 MB of space.

Then, of course, there are the inevitable failures of this network machine with its many moving parts. Murphy's law kicked in repeatedly, resulting in about a quarter of the computers in the GAM computer lab failing. When you have 5 separate people logging on to each machine each day, it thrashes the hard drive, which can lead to drive failure. Typically this occurs when someone is using the machine, possibly in the middle of something important...

The network itself had multiple outages, repeatedly disrupting our workflow. The school did what they could to keep the SVN running, but even that crashed. It is probably not a terrible idea to invest in a remote code repository. It also might be worth exploring GIT as an alternative to SVN.

GUI Issues – In the last four months of development, the GUI programmer (Alex) was faced with the challenge of fixing the interface problems related to resolution changing and adding more interactive controls (sliders, radio buttons). The logic behind interface objects was in the game state, which made coordinating interface changes difficult. The logic behind these elements was eventually moved to the interface object classes. To avoid excessive coupling of the Model and the View, messages were passed in when constructing the elements. These messages would be sent by the interface manager and received and processed by other game modules.

Design Overview

What Went Right

The Interface – Alex spent a lot of time creating and then re-factoring the interface objects. During the early stages of the game, the interface was very blocky and unappealing. Through continual feedback from both the GAM professors and the team, Alex continued to tweak and change the interface, making each piece fit while compensating for resolution changes. By the end of the project, we had a beautiful interface with good feedback elements, providing an overall positive user experience.

The Particle System – Tommy's particle emitters worked well and were pretty flexible. It was easy to extend and build a variety of particle effects. At first, the fact that all particles were in the same texture file was a bit odd, and created something of a design challenge. However, since Thom had created a set of explosion textures in a single texture previously, he was able to recycle the large number of textures within the file (combined with color changes and animation) to create new effects. Feedback on the particles and explosions was positive, and it contributed greatly to the graphical polish.

The Mechs – Although we did not gain any artist-created assets beyond Roberto's buildings/environmental textures, Thom's mechs added a unique style to the game. Without the mechs, it would have been very difficult to pass our game off as a “mech shooter.”

The Controls – Through a lot of playtesting, we ended up with a very refined set of controls. Originally dashing was controlled by double-tapping the directional buttons (WASD). Play testers didn't like this, so a modifier key was used instead (shift). Later, Alex added the ability to change mouse sensitivity and map custom controls.

The Height Map – While it wasn't the best behaved aspect of our levels, it did allow us to easily create aesthetically-pleasing mountainous (Martian) terrain. Unfortunately, because of its limitations (proper collision with walls, etc), it was only used to great effect in level 2.

What Went Wrong

Design changes forced by artist schedules/moratorium – Again, the lack of artists really affected design decisions. We had already dropped the cutscenes when the artists told us they were overloaded, but the moratorium forced us to drop flying mechs, among other elements. This forced game design changes, and the last remnants of our back-story were cut from the game.

Not Enough Playtesting – We did do quite a bit of playtesting early on, but in the last couple weeks of development, we didn't have enough time. At the same time, a lot of elements were added, increasing the the challenge and iterating on the core fun factor (blowing up mechs). These elements were largely untested at the time of our Gold release, and we have some potential balancing issues.

AI Put Off Too Long – Near the end of the development cycle, Thom created a passable A* pathfinding system using a navigation graph. However, this came too late to become integrated in our level editor. A basic state machine system (for game objects) had been implemented near the beginning of the development cycle, but suffered architectural issues. This root of this problem stems yet again from bottlenecks encountered by small teams. Thom was also responsible for creating a resource manager, which accounted for all the resource loading/accessing/unloading functionality. This took up four months of the eight month development cycle, and required maintenance beyond its initial creation.

Height Map Limitations – The height map was never as flexible as it was originally supposed to be. There was no accounting for the slope of the map, so if a player is walking into the side of a cliff, it will propel them to the top. This was a huge pain to deal with for the second (canyon) level. Thom had to place invisible collision boxes all around the walls (through the level editor) to prevent the player from physically running into them and getting pushed to the top of the map, possibly causing a sequence break. This also forced us to change the design of the third level. Originally the city was planned to have tiered plateaus leading up to a central plaza, but the height map didn't handle the ramps or hard-edged cliffs very well, so we redesigned it to be completely flat.

Level Editor Limitations – While it was easy to add, remove, and change the position of objects we could not rotate or scale them. Scaling probably wouldn't be that useful (it's set in the pattern for the object, not the orientation information), but the lack of rotation was a huge pain. A whole new object type had to be created for each separate rotation of an object. This also meant that all our mechs start off facing the same direction.

The Mini-Map – Because the design had the mini map as being circular, it made clipping things to it hard without using shaders (which though originally planned, would have required a good amount of time to implement when taking resolution changes into account). In the end, Alex engineered the mini-map so that objects fade out as they get further from the center. This method was aesthetically pleasing, barring a few cases with long walls/large objects.

Final Libraries and Pre-Built Tools List

The following is copied from the final version of the TDD and lists the libraries/tools used to make the final game.

Game Libraries

OpenGL	Rendering. Version: OpenGL 2.0 with GLSL shader model 2.0 (at minimum, optimized for 4.0)
GLEW	OpenGL extension management. Version: glew 1.5.1
GLFW	Originally used for window management/input, now only used for the high-res timer. Version: glfw 2.6
SDL	Window management, input, resolution changing. Version: SDL 1.2.14
FMOD	Sound. Version: fmod 3.75
Lua	Scripting. Version: Lua 5.1.4 (August 2008)
Boost	Containers, synchronization (threads, hash tables, etc). Version: Boost 1.37.0 or later (up to Boost 1.40.0 works so far)
TinyXML	XML serialization. Version: TinyXML 2.5.3
SOIL	Image/texture loading. Version: July 7, 2008 release. http://www.lonesock.net/soil.html

Tool Libraries

Autodesk FBX SDK	Used for building tools to generate model/scene files from FBX files. Version: FBX SDK 2010.0 and 2010.2
wxWidgets	GUI for tools. Version: wxWidgets 2.8.10
Visual Leak Detector	Better output for Visual Studio's built-in leak checking. Used in Debug. Version: VLD 1.0 (August 5, 2005)

Tools

SWIG	Generating Lua glue functions. Version: swig 1.3.36
Autodesk 3DS MAX	Animation and modelling. Version: Autodesk 3DS MAX 2010 (12.0)
GIMP, Photoshop	Textures, logos, etc. Version: GIMP 2.0 and above. Photoshop Elements.
MS Visual Studio 2005	IDE/compiler. Version: Microsoft Visual Studio 2005 (IDE Version 8.0.50727.762) Command line compiler version 14.00.50727.762
Cygwin Bash	Automation for various tasks. Mainly used for building the installer. Version: GNU bash, version 3.2.x or later for Cygwin
NSIS	The installer compiler. Version: Nullsoft Scriptable Install System 2.24
AMD CodeAnalyst	Used to identify performance bottlenecks. Version: AMD CodeAnalyst version 2.9

Old Libraries and Tools (not used anymore)

TiCpp	XML serialization. Dropped because it is broken under MSVC 2005. Version: ticpp 2.5.3
Cygwin Make	Scripted builds. Dropped after Makefiles were not properly maintained. Version: GNU Make 3.81 or later for Cygwin