# The Genesistrine Network System

A classroom system for teaching 3D computer game technology
Dave Anderson – June 19, 2009
Preliminary – a work in progress

**About the Name**

Charles Fort (1874 – 1932) was an American writer who chronicled “anomalous” phenomena throughout the world. The classic Fortean event was a rain of frogs, fish, or other peculiar stuff falling out of the sky for reasons unknown. He offered explanations ranging from the mundane to the occult, the paranormal, or the just plain weird.

Fort proposed (undoubtedly somewhat tongue-in-cheek) that, somewhere above the Earth, there existed an invisible continent populated with a wide variety of unusual life forms which periodically spilled over and rained down on Earth. He called this continent Genesistrine (first articulated in *The Book of the Damned*, 1919, p. 73), “knowing no more what it’s all about than we do when we crawl to work in the morning and hop away at night.”

With homage to Fort’s love of weird phenomena and scepticism towards all rational explanations of everything, “Genesistrine” is the working title for my educational game environment, where anything can happen and need not be explained. And, like Fort himself, skeptical of authority and always subject to change.

**An Overview**

Genesistrine consists of a number of components aimed at creating an environment for teaching game engine technology, arts and crafts. A large part of Genesistrine is built around the Torque game engine (widely used in the academic world and published by GarageGames.com), and consist of a number of “worlds” offering educational content and training wheels to build creative works within the Torque framework.

Genesistrine combines teaching and gameplay in a fairly unified manner. The student starts out in a game world with virtually nothing in it – a flat floor, a light source overhead, and a teleporter for visiting a number of “resource” worlds. Each of these resource worlds is focused on one aspect of 3D game technology – terrain, objects, textures, water, buildings, environmental effects (sun, rain, snow, lightning, sandstorms), particle effects (fire, explosions), weapons, vehicles, avatar bodies, etc.

Each of these worlds has a test associated with it – in order to bring assets back to their home world, the student must pass the test. Sometimes the test is to build their own things, while other worlds may simply require them to answer questions. Quizzes could be built in, while constructing things may require the instructor to sign off on the effort. The home world teleporter will only allow students to visit certain worlds, depending on which tests have been successfully passed – sort of a merit badge system.

Each student must construct (or acquire) everything in his or her own home world. These home worlds would be quite individual. Students will be able to visit other student worlds at appropriate times. Hopefully, the more advanced students could help others in the process.

These home worlds should probably not be called "games," although there will be game elements in them. It's unreasonable to expect a student to construct a real game in one semester with no prior experience. I prefer to think of them as "toy" worlds, although there can be puzzles to solve, enemies to shoot and run from, and things to do which will give the player points.

One problem I can see is the amount of resource files the system will require. This would require assistance from a number of people who are skilled in such things. There are some free resources here and there on the Internet, but this won't cover everything that's needed.

Underlying much of what Genesistrine does is a networking system that allows students to:

- work on their projects from various locations (school, home, coffee shop);
- access common resources (Flash drive, teacher's computer, or server);
- save and load their work to and from various places (like a Flash drive);
- revert to a previous day and time in the event of a mistake;
- submit their work as homework or project completion;
- collaborate with one another when allowed by the teacher;
- and play each others' games, or otherwise evaluate class work.

Although I've described the Torque game engine as the system of choice throughout much of the rest of this project description, in the networking component the choice of a game engine is largely irrelevant. Torque offers superb network functions; but much of the following could be implemented in Beanshell, the Java-like scripting language for the jEdit editor (which is presumed present and operating for most of the Genesistrine functions), or Python (another excellent choice of scripting language for networking functions). Beanshell gives easy access to any function available in the standard Java libraries, and has the advantage of being easy to program for most of the functions in the Genesistrine networking component. The main difference is playing (or visiting, i.e. walking around in) other students' game worlds, which can be handled best by Torque. I wouldn't presume to supply this function for other game systems.

jEdit is a good programmer's editor, and is open source, meaning no licensing problems for a Genesistrine plug-in. It runs on Windows, OS X, or Linux without any appreciable difference. Changing the jEdit editor code itself would require releasing the code without charge (as per the GNU licensing); but writing a plug-in for jEdit can be done commercially, and that's the approach taken here. JEdit supports a large number of programming languages and compilers. Stubouts (i.e., software connections) can be provided for most unsupported programming languages that adhere to POSIX standards.

## Some System Components

**CVS**. One precedent to the basic Genesistrine network component is the CVS system used by just about all programmers who collaborate with others (and even many solitary programmers). CVS stands for "concurrent versioning system" – a networked client-server code library system which manages changes to some large shared body of code. There are numerous implementations of the CVS concept, but they all basically work alike.

In practice, the programmer starts a CVS client and connects with a server to acquire some portion of the current project code; he or she writes and tests particular changes; and then checks the changed code back into the common code library once it passes muster. Students, however, aren't generally collaborating on one body of code, but rather creating their own variations, with a basic code body as merely a starting point. And we don't want students to access each others' work without explicit permission. Further, there is a particular type of "check-in" procedure needed here to identify a student's submitted homework, or solution to a class project requirement.

The typical game development project contains lots of files across many directories. A CVS system stores only those files which have changed. Turning in class projects wouldn't shuffle around huge amounts of data, but only those files and directories which are different from the original materials that were shared among all students at the start of the class or project. Common in CVS systems is some kind of "difference" utility, which finds and points out files (and lines of code within files) that changed from one save to another, which will assist teachers in helping students debug their work.

**MAKE**. Another precedent for Genesistrine is the universal programmer's utility called "make." Make is a tool that parses build scripts. Each script declares a number of "target": states for the system, and describes functions that need to be performed on the various files in order to achieve that state. In the usual programming situation, these functions are usually invocations of the compiler, turning code files into binary components of the whole project, and binding them together. Genesistrine expands that model by defining "phases" of the course that can be brought about on all of the class computers. The beginning state of a class project, for example, can be brought about by making new resources available on the students' computers, changing the default behavior of various applications and utilities, or searching out and acquiring files from the students' machines to satisfy some homework requirement. It's very much like the standard make utility, but with a master-slave sense built in (allowing the teacher to do things that the student's can't), a time sense built in to handle various predefined phases of classroom work; and with the ability to work across a number of networked machines.

In practice, the Genesistrine make utility performs some functions immediately (such as on-screen tests and quizzes), and pushes scripts for other functions to the students' machines (actually, to their Flash drives) that present new capabilities (such as turning in homework assignments). This looks like a dialog panel on jEdit that presents a set of functions which can be activated by the student by clicking on things, or optionally by typing in commands in a text window.

Tests, for example, could be handled as HTML files, activating the Firefox browser on any active student computer, and feeding it a web page from the teacher's computer.

## Smart Classroom Software Avoided

There are some features of the ideal classroom network system that may already be in place in today's "smart" classroom systems. These features include allowing the teacher to see what's on the screen of any particular student's computer; monitoring executing software and Internet connections; Internet co-browsing; and so forth. Right now, I don't see a need to specifically address such things. I don't want to duplicate functions that aren't essential, and may already be provided by others. There are certain overlapping functions, however, such as turning in homework, which will be different in Genesistrine. In the game development context, students should only turn in files which they have created or modified from the originals. E-mail and instant messaging are not addressed either – both potentially useful, but there are plenty of other alternatives. Note that e-mail isn't an acceptable means of turning in most homework in this context.

There are difficulties in having student software and resources located in different places – for example, a hard drive and a Flash drive. To eliminate these problems, it is assumed that the current project and all executable software is on the hard drive of the student's computer. I also assume that the hard drive is reinitialized to some basic condition when the computer is rebooted. This is a common situation – each student computer is reset to some basic state between classes, and in the event of a crash. So, the basic practice here is to 1) boot the computer; 2) load the current project state onto the computer's hard drive, and 3) execute all software and resources from the hard drive. The student may create multiple versions of a project on a Flash drive; but this creates an intolerable situation for the teacher to debug when needed.

## Some Basic Software Engineering

I suggest that one of the basic components of any game development class is to know core software engineering concepts. A computer project consists of a number of files. Those files are located in a number of directories, which have a particular structure. Progress in the course requires modifying existing files and creating new ones, as well as modifying the structure of directories that hold these files. We can help students find files and directories; but understanding the architecture of files and directories is an essential cognitive construct in the art and science of building software systems.

A quick and dirty definition: When you're putting code into one file, you're programming; but when you're putting interdependent code into two or more files, you're engaged in software engineering. Debugging isn't just finding errors in some file of code; it's understanding the differences between versions of code files, and the combinations thereof. In debugging, something changed. It was running, now it isn't. What changed? Where? When? How do you find the problem? A game technology course is a relatively easy way to introduce these concepts, and instructors shouldn't avoid them.

I will also include the possibility of a centralized server for storing student work. A central server is certainly appealing for periodic backups, and possibly creating one common class game system. However, for doing homework, the Flash drive storage scheme seems more promising – students can do their homework without accessing the school's servers. The teacher should be able to make certain choices about the storage of basic class materials when setting up the system for a particular class, including specifying a server rather than (or in addition to) a personal Flash drive. Indeed, this whole system would work without Flash drives. But the Flash drive solution 1) identifies each student upon insertion; 2) simplifies porting class work to home computers; 3) avoids the problem of tampering or accidental damage by other students (without depending on encryption and other security measures); and 4) avoids clogging the LAN with unnecessary traffic.

I'm assuming that for most situations, security considerations are fairly light. Student work will be protected by a password, but tighter security measures probably aren't needed. Only the teacher will be able to access the teacher's computer's repository in write mode. Likewise, only certain student files can be changed by other students in collaboration mode, and there will be a "revert" function in case the collaborator messes things up. But I see this kind of thing as a fairly routine set of security measures. In other words, NASA would find my security measures unacceptable; but most teachers don't have to worry about skilled hackers.

In the system description that follows, I frequently refer to storage on student and teacher Flash drives. Wherever possible, I have attempted to avoid assuming that there is a central server.

## Basic Setup

There are a number of computers in a class, all having access to a network. These are presumed to be hardwired together, although that may not always be the case. Local laptop computers could be attached to the class through wi-fi. And some (hopefully all) students will be working at home. It is quite desirable to accommodate these situations, although I won't flesh them out for a first pass at the networking system. The baseline here is a number of computers on a LAN – everything else will be reserved for later.

There is a basic, identical set of resources on each computer. These will be set up at the beginning of a semester through the usual classroom "cloning" software (such as Norton Ghost), which need not be supplied by Genesistrine (although it could be very easily). It is presumed that these basic resources (files and directory structures) are quite large, hence their presence on each student's hard drive.

Students will set up home computers with the same basic resources, either by doing a "save" to their Flash drives (which, when nothing is on the Flash drives, will save everything) and then restoring to their home machine; or by using a DVD or other disk supplied by the teacher or included with some text.

One computer is the teacher's machine, which has additional resources. It is not assumed, however, that the teacher is present at each class meeting – only that the teacher's computer is turned on. Such additional resources may be on some other server, at the teacher's discretion.

Each student, and the teacher, are identified by a Flash drive. Guests may attend a class without an identifying Flash drive, and have access to the basic materials. Basic materials may be acquired from the teacher's computer, or some other server, although there will probably be a penalty in downloading time. Likewise a student who forgets his or her Flash drive will have some other alternative to storing the day's work. I've had success in offering one class "loaner" Flash drive to forgetful students, although there is a question as to how the student's daily work gets to his or her home computer for doing homework, since the teacher would retain this Flash drive between classes. This is a (minor) issue to be resolved.

Students may move from one computer to another without any problem. It is not presumed that a student always uses the same class computer.

Differences (and only the differences) between the basic resources and each student's developed resources (in other words, their ongoing work) are stored on the Flash drive. Likewise, differences in directory architecture will be saved. These differences are called the student "delta files."

"Load" and "Save" functions are provided which move the student's project delta files to and from the student's Flash drive. It is assumed that students do at least one save, at the end of a class. They may do any number of saves, and the system will record both the date and the time. Optionally, a label may be applied to a save, such as "finished project 1." Load, by default, brings the last work onto the current computer, although the student may resurrect any previous date-time save, or a label save.

There is a facility for the teacher to define particular class work. A "turn-in" function is provided for each student to move his or her delta files to the teacher's machine. It is assumed that these delta file sets are stored on the teacher's Flash drive, although they could, at the teacher's discretion, be saved on some central server.

There is a sense of history in the system. Using the CVS system, students may revert to some previous date and time if they find they have made mistakes.