Cannibal Island

Final Report

Ben Tietgen, Jared Hoberock, Dan Kelly, Neil Sawford

Mentor: Dr. K. Palaniappan

May 10, 2002

1. Executive Summary

"Cannibal Island" is a struggle for survival in an unfriendly environment. This video game will place the player in the role of a shipwreck survivor struggling to survive on a deserted island. With the use of sophisticated audio analysis and an advanced three-dimensional game engine, the combination of digital music with the fascinating world of computer gaming can become reality. By using the unique patterns within the music to determine components of the game environment and activity during play, computer gaming will become a musical experience.

Music has played an important role in every human society since the beginning of time. Music has been written to celebrate, to provoke emotions, to entertain, and to tell stories. In "Cannibal Island," the player will select a song from his or her digital music collection. This song will help determine the "story" to be told. The terrain of the deserted island will be determined by aspects of the song, such as tempo, dynamics, beat, key, and rhythm. Fires will flicker and trees will sway to the beat of the selected song. Computer-controlled agents will react to highs and lows of the song. The possibilities are endless. One thing is certain, each song will generate a unique island and help the player tell a different story every time the game is played.

Video gaming became popular in the late 20th century and is still popular today. There are many types of games being played today, such as "Role Playing Game", "First-Person-Shooter", and "Third-Person Shooter". These types of games require the player to take on the role of the game's main character and attempt to win the game. In a "First-Person-Shooter," the player is viewing the game as through the main character's eyes, a first-person view. A "Third-Person-Shooter" style game places a camera above and behind the main character so the player can see the main character's body and his movements throughout the game, a third-person view. A "Role Playing Game" typically uses the third person view. Also, many action games give the

user the option of playing in either the first- or the third-person view. "Cannibal Island" will give the player the option of using either the first- or the third-person views.

Video games and digital music have never been combined like this before. This project is an innovative attempt to bring these two phenomena together. Although this project will be tough, it can be completed efficiently and at a very low cost. Video game and digital music fans will definitely meet this novel venture with approval.

2. Problem Statement

2.1 Background

Our culture's fascination with video games continues to reach new levels as the market presents consumers with more choices than ever before. Currently, gamers may choose among three different next generation consoles, one sponsored by Microsoft, the undisputed king of the digital world. Game programming, once the esoteric hobby of a few hackers 20 years ago, is now considered big business.

The digital music phenomenon also continues to gain momentum. While still arguably an infant technology, the acronym 'mp3' is well ingrained in pop culture. As the legal battles and business deals rage in Cyberspace, computer-savvy music lovers continue to download their favorite songs on demand.

Our project hopes to combine these two phenomena in an unprecedented way. Imagine a game environment governed completely by your Winamp play list. Imagine computer-controlled opponents that react to the highs and lows of your favorite song. Imagine a game whose replayability is bounded only by your music collection. These are just some of the ideas our team will explore in our senior capstone project, tentatively titled "Cannibal Island."

The algorithms necessary for music recognition are just now beginning to emerge. Independent researchers have recently begun to develop intelligent software to recognize the basics of musical audio, such as tempo, and rhythm [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. In a few cases, some of these algorithms are robust enough to recognize genres and partition songs into musical movements [11], [12]. We hope to combine these algorithms in such a manner as to create a music-responsive game environment.

2.2 Goals & Objectives

Figure 1: The Torque Games Engine in action

2.2.1 Premise and Game Environment

Our group hopes to place the player in the role of a shipwreck survivor struggling to survive on a deserted island (see Figure 1). The game should provide for full interactivity with the island environment, from swimming through pools, to climbing trees, to lighting campfires. A 'third-person' camera (see Figure 4) will follow the player's shoulder as he runs, jumps, and climbs through the environment. The game will be graphically impressive, complete with sparkling water effects, menacing rain clouds, and verdant jungle foliage. The game will feature characteristics popular in most third-person action games. Game situations will include running, jumping, collecting items, avoiding enemies, and exploring the island.

2.2.2 Audio Recognition

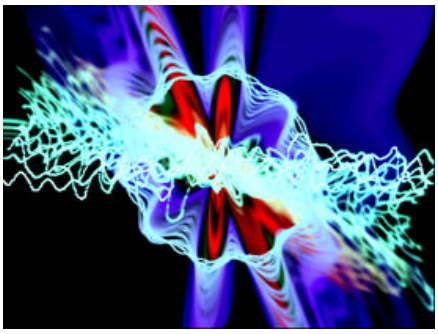


Figure 2: Winamp audio visualization

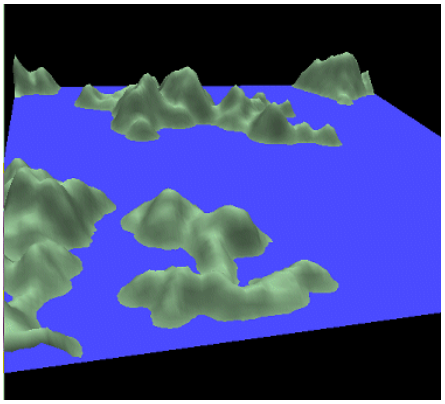


Figure 3: Synthetic terrain

The clever implementation of audio recognition will provide for a novel gaming experience. We hope for both the physical geometry of the island, and the behavior of in-game physics to reflect the music choices of the player. We believe the current state of musical audio processing will be "smart" enough to recognize the musical features to which the game should respond [12]. These algorithms will process digital audio information both in real-time and offline.

Offline processing should generate the geometry of a unique terrain (the island) for a unique song or collection of music (a play list). Current music visualization software does little to illustrate the unique character of a song (Figure 2)[13]. Our team hopes to employ cutting-edge music recognition algorithms to extract features such as tempo and rhythm to produce appropriate terrain. For example, a slow ballad could correspond to rolling hills while a fast rocker could produce a jagged mountain range. Current synthetic terrain generation algorithms are well established and provide convincing results [14], [15], [16]. We hope to "seed" these algorithms with appropriate parameters provided by our recognition algorithms to produce a corresponding terrain. Figure 1 depicts synthetic terrain generated by the "particle deposition" algorithm.

Real time processing should provide music information for in-game events as the music plays. Faster, more crude algorithms will provide a stream of music data to the game engine so that events and artificial intelligences can respond to musical events. In this way, the player will influence the game world with his selections of music in real time. For example, palm trees might sway to the beat of a song, or an angry animal could be scared away by the loud sound of a drum. A campfire's flames might be keyed to the flame-like spectral data of a Fourier Transform. Camera cuts could be keyed to salient note onset events in a rapid-fire MTV music video style. Ominous clouds could respond to a song in a minor key. The possibilities are endless.

Fortunately, the beginning of a robust music-recognition library actually exists as open source. This code, dubbed "MARSYAS," exists as a framework for fast prototyping of music-recognition algorithms, and has already been used to implement song-recognition for databasing and retrieval. The library boasts such tools as beat extractors, Fourier and Wavelet Transform extractors, and, most impressively, a segmentor, which actually parses songs into verse-chorus-verse partitions [12]. We hope to adapt the code in this library to provide intelligent recognition for our implementation.

2.2.3 Graphics

Three-dimensional video games, popularized by Quake and its many imitators, continue to be the genre of choice of today's gamers. In fact, three-dimensional visualization is the driving force behind video game console developers and computer graphics card vendors, as new games continue to push polygon counts to the limit. Our game will be no exception. We hope for a professional-quality, visually appealing experience.

Our team will employ a professional game engine for the high performance visuals our game requires. A clone of the Quake 3 engine, arguably the most visually impressive 3D computer game ever produced, is available as open source. Another proven engine, the "Torque Games" engine (Figure 1), is available from www.garagegames.com, complete with documentation and tutorials, for only $100 to developers. This particular engine lists features such as an artificial intelligence engine, a particle engine, a 3D engine, and development tools, perfect for a game like ours [17].

Figure 4: Super Mario 64

Figure 5: Winamp player

With the help of our selected game engine, we will develop a unique user interface to provide both responsive control of the player character and quick access to the user's digital music library. Ideally, a third-person game such as ours should employ an interface similar to "Super Mario 64", the archetypical third-person action game [18]. However, the absence of a control pad and the addition of music access complicate matters. An appropriate solution must allow for both Winamp-like music access and hair-trigger player responsiveness.

2.3 Overall Approach

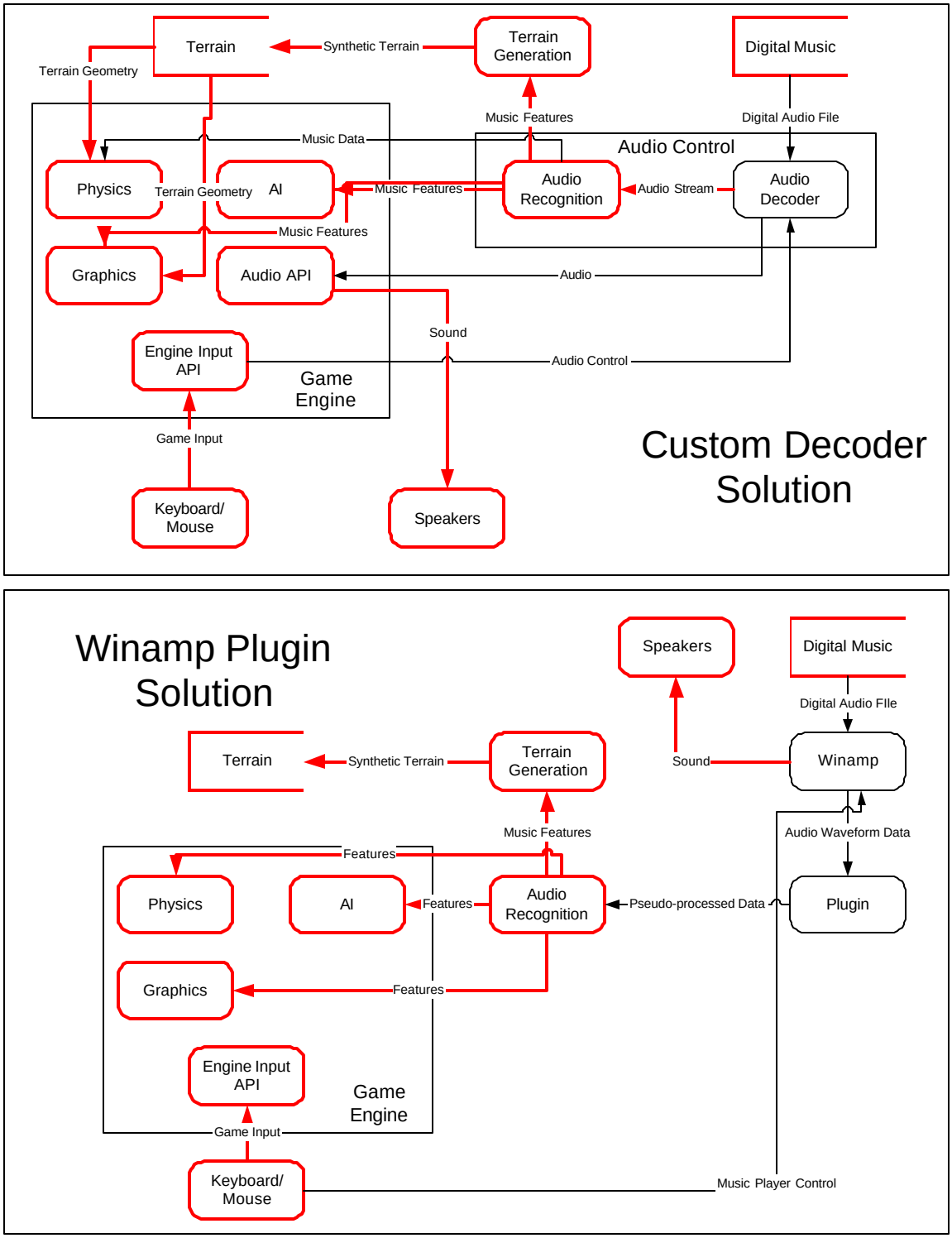


Figure 6: Two proposed high-level system implementations

2.3.1 Prototyping

We hope to divide our efforts simultaneously among our project's three most visible components: music recognition, game engine implementation, and user interface (See Figure 6). The prototyping of these components will proceed in parallel. For instance, synthetic terrain generation from music need not depend on the actual graphics implementation. Therefore, graphics prototyping and audio recognition prototyping can occur simultaneously for final integration later. Once we overcome these initial hurdles, final tweaking and "bells and whistles" will be trivial by comparison. We hope to have several prototypes for these components implemented by the end of the semester, leaving the winter semester for integration.

Prototyping for the music recognition component will involve both real-time and offline processing. Since both methods use the same recognition library and employ similar algorithms, it is only logical that we group these two tasks together. Terrain-generation prototypes should illustrate how different musical features affect synthetic terrain-generation. Throughout the prototyping process, appropriate choices will be made with respect to what musical features correspond to what terrain features (slow songs produce smooth terrain, fast produce coarse terrain, etc).

Since audio recognition is such a computationally expensive problem, real-time processing will no doubt bound the performance of our game. Some compromise will have to be made between precision and performance. Real-time processing prototypes should favor speed over accuracy in order to provide data to in-game events. A recent paper on beat detection suggests that good recognition is possible at only 15 Hz [9], [10]. This gives some indication at how often we must sample and process our music for accuracy. Given the performance of this component, we determine which musical features will map to which in-game physical processes and events (loud songs produce aggressive enemy AIs, etc).

Initial graphics prototypes should demonstrate the "look and feel" of the game. Once we solve the basic problem of instantiating a character in a 3D world with our selected game engine, we can experiment with user interfaces, visual embellishments, and game physics. We feel that a professionally developed game engine will facilitate this kind of rapid, dynamic experimentation.

The actual decoding and playing of digital audio in-game presents a unique problem. For example, the popular digital music player, Winamp, offers developers an interface for extending Winamp's functionality. It is entirely conceivable that a plugin could provide our main program the audio information it needs as Winamp plays our soundtrack in the background [13], [19]. This would circumvent the issue of writing a custom mp3 player, processor, and user interface. At the same time, it would extend our game's functionality by allowing alternate audio sources, such as Windows Media Format, compact discs, or most excitingly, microphone.

However, developing a Winamp plugin may bind us to legal issues and will also fragment the user interface into both a music and play-control interface. Ideally, game control should be cohesive throughout, with no dichotomy. There is an alternative. Mpg123, an open-source mp3 player project, exists, which could be adapted to our game [20]. However, this will no doubt limit the variety of audio formats for the player to choose from. This is another decision that a variety of prototypes will help evaluate.

2.3.2 Development Tools and Languages

We will develop our game for the Win98/2000 platform. The system requirements for *Tribes 2*, a professionally produced game based on the Torque engine, recommend a 400 Mhz Pentium II PC with 64 megabytes of RAM [21]. We feel

Figure 7: Tribes 2 screenshot

this is an appropriate target platform for our game.

Development will proceed on a variety of platforms, depending on the need and convenience. We will utilize both PCs and Silicon Graphics workstations on and off MU campus. We will produce code under a C/C++ framework, using object-oriented design whenever possible. Code will be compiled and built using PC-based MS Visual C++ and Unix-based GNU tools, both of which are available on campus. In addition systems analysis and configuration management will be facilitated through Visio, Rational Rose, and other software engineering tools.

2.3.3 Resources

The only tangible monetary expense incurred by development will be the purchase of the game engine. This is likely to be the Torque engine, which GarageGames offers for $100. Conceivably, the MU campus computing fee could be applied to our project, which is $8.90 per credit hour [22]. Senior Capstone Design is a four-hour sequence, which, multiplied by four team members at $8.90 per credit hour, comes to $142.40. When one considers the capabilities of the workstations and PCs available on campus, this is a bargain. Of course, facilities at a professional programming studio would be more expensive.

We can only estimate other costs. We feel game development will demand at least 50 man-hours per week once prototyping begins in earnest. Other costs associated with distribution of the final product, such as providing open source code, are more difficult to analyze.

3. Requirements Analysis

3.1 Constraints

3.1.1 System Components Described

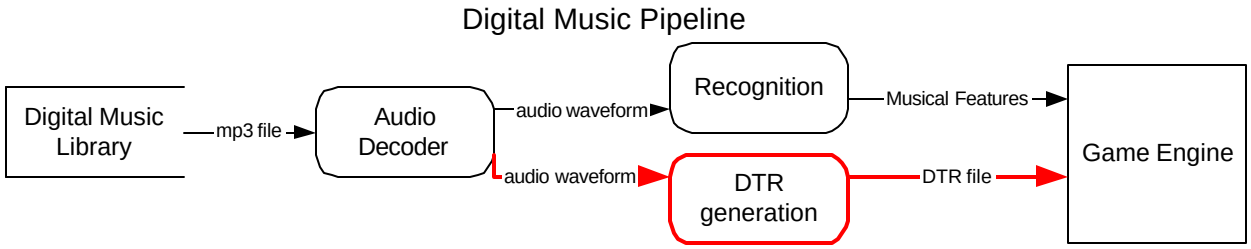


Figure 8: Digital Music Pipeline

The goal of the audio decoder is to take a digital audio file, in any format we decide to support (current plans are to support mp3 and wav files), and convert it to a format where it can be analyzed for its various musical components. Digital conversion algorithms used by programs such as FreeAmp, a digital music player, will aid in the development of our own algorithms. The resulting formatted file will then be fed through the pipeline, first to the terrain generation program, and then to the real-time event synchronizer, which only runs while the game is being played.

The terrain generator requires a formatted file created by the audio decoder. It extracts frequency information from the file using algorithms based on the MARSYAS filter programs. It manipulates the frequency data to create a unique map of the sound file, which can be interpreted by the game engine to create the in-game environment. Data in this file includes elevation, water coverage, climate, weather patterns, plant and animal abundance and growth rates, hostile presence, available items, and game length.

With real-time music processing and recognition, we hope to simulate (to a crude approximation) the human perception of music. As music plays, real-time processing and

recognition will provide a constant stream of interpreted music data to the game engine to influence game events and behavior. The processor algorithms will utilize the decoded sound file to detect musical features while the sound is playing. Current research suggests that tempo, beat, and dynamics can all be precisely recognized in real time.

We decided that the most viable solution for a game engine is to learn an advanced, open-source commercial engine. We chose the Torque game engine, used in the video game *Tribes 2*, because it is a full-featured game engine. Torque consists of the platform layer, scripting engine, GUI engine, mission editor, 3D engine, mesh engine, particle engine, terrain engine, interior/building engine, water engine, networking, sound, and various other tools and libraries. Understanding this engine is vital to the success of our project, as we intend to implement additional features which may complicate the code, such as growing, climbable trees, music-affected weather, and the ability to manipulate the terrain (i.e. dig holes).

3.1.2 Implementation

Digital music conversion is not a new topic; programs such as FreeAmp and MARSYAS take the digital information in a file and drop it into a compressed buffer. Interpreting the buffer is simple, but time-intensive. To provide our game with the ability to recognize wav and mp3 files, we can covert MARSYAS from a Linux to a Windows environment. Since MARSYAS already recognizes the wav file format, we must only extend it to recognize mp3s. This can be done by analyzing the algorithms used by FreeAmp, which recognizes mp3s. Once we can decode digital music, we need only to export the audio file in a format recognized by the terrain generator.

To build the terrain generator, we must first decide what audio data we wish to represent. Frequency data inherent in audio information is the most logical choice. When frequency data is

combined with song progression, a visual representation already resembles terrain [Figure 9]. A height map for the island can be produced with the raw frequency data, or by seeding a randomizer function with the data. The height map should be in the form of a matrix that the game engine is able to read. In addition, game aspects such as climate and vegetation can be determined by analyzing frequency data. For example, if we list the possible climate zones as arctic, temperate, tropical, and arid, we would assign each zone an enumerated integer value. Then, we could extract the time in the sound file where the first zero crossing occurs and seed a randomizer with that value, modulating the result to determine the climate of the island. The height map and extra data is stored in a digital terrain representation (DTR) file. The result is an executable that takes a sound file or playlist as an argument.

Figure 9: frequency analysis

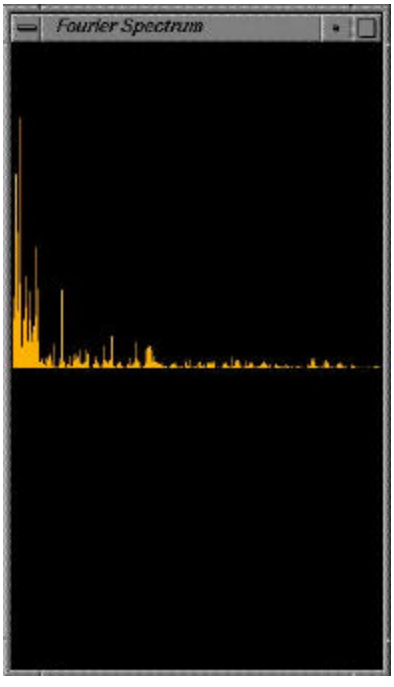


Figure 10: prototype in action

Beat detection seems to be a relatively simple problem in musical recognition. We have identified two candidates for algorithm implementation [9], [12]. Both candidates use digital signal processing approaches to perform beat detection, as opposed to differing models that employ finite state automata [4]. This similarity will facilitate quick prototyping and evaluation. Beat detection prototypes [Figure 10] will be implemented using the MARSYAS audio processing library classes. The implementation of our two other musical features we wish to recognize will be somewhat simpler. Since tempo

detection naturally depends on beat detection, beat detection prototypes will be quickly extended to recognize tempo. Specifically, beat frequency will define tempo. Dynamic recognition will be facilitated through tracking changes in RMS amplitude of incoming waveforms at an appropriate sampling frequency. Since musical dynamics change on a scale on the order of seconds, an appropriate sampling period will reflect this observation.

Modification to the game engine should be made in a modular fashion in order to keep track of performance issues and to provide suitable testing environments. No code additions or changes will be made until the team has attained a sufficient level of understanding of the engine's features. Since the engine is written in C++, all enhancements will also be implemented using C++.

3.2 Performance Requirements

The audio decoder will be implemented to work on a computer with Microsoft Windows. The user of this system will be the game-player. Source code for MARSYAS and FreeAmp will be used to aid in construction. For this part of the game, a computer with at least an Intel Pentium processor running Windows 95, 98, 2000, or NT 4 is required.

The terrain generator is bounded by predetermined terrain data. We call it the "audio environment" – the generation of terrain is seeded with information from a specific audio file. Ideally, this means that each sound file defines its own unique island world. The user will never need to deal with the terrain generation code; it is self-contained and it relies only on a viable formatted file received from the decoder. The execution time of the generator is dependent on the amount of data (i.e. length of a song) to be interpreted. The code for the terrain generator will be written in standard C++ for a Windows PC. In addition, we will reference the MARSYAS code and algorithms often.

The real-time processor is the most visible aspect of our music recognition algorithms to the player. Any inaccuracies in system performance will be immediately recognizable to the player. While terrain generation from music will be somewhat subjective, by contrast, real-time analysis of musical events must be precise. For instance, there should be no question as to when a particular musical event, like a tempo change, occurs. For these reasons, the team feels that a successful system solution should aim to recognize just a few real-time musical features, and to recognize them precisely. We will implement additional recognition features (i.e. musical key, musical salience, and musical segmentation) as performance allows.

The Torque game engine documentation lists the system requirements as follows:

```
Win 98/2000 platform
400 Mhz Pentium II PC with 64 megs RAM (target system)
```

We aim for a frame rate of 30 fps and instantaneous response to user-triggered events on the above machine. We expect the performance of the game to slow as we add features to the engine, and we will solve those problems as they arise.

3.3 Cost Requirements

Overall, the monetary costs of the project are minimal. Access to the game engine and documentation costs $100. All other programs utilized are open-source. In terms of man hours, each aspect of development is expected to require between 50 and 100 hours of design, implementation, and testing. The audio analysis software will be developed in the SGI computer lab on the University campus. The game engine modifications, including a redesigned user interface, weather system control, and character and entity modeling, will be done on our personal Windows systems.

3.4 Alternative Design/Solutions

Regarding our decoding solution, it is possible to use WinAmp plugins to get the mp3 information into the correct format to produce a world map. However, the WinAmp decoding format does not provide all the options that our solution does. There also may be legal issues with America Online for the use of WinAmp, one of its products, in the game.

We have determined that the matrix data format for the DTR file is most efficient in terms of generation speed and space conservation. In addition, many commercial games have complicated map formats because of the different types of environments they must represent. Our game does not need to interpret data for buildings or other physical structures, so we will be able to streamline the interpretation algorithms built into the game engine. Thus, our solution also shortens the amount of time necessary to generate and interpret map data. Therefore, no alternative designs should be considered.

After substantial research, we have been unable to find an audio processing library comparable to MARSYAS. The only alternative to using MARSYAS that we have considered is to start from scratch and write our own library. The advantage to doing this is that we will know what analysis each function in the library performs, inside and out. However, we would rather spend half the time necessary to do this and use it to learn MARSYAS as completely as possible.

There are many different game engines available that could serve as the basis for our game. A very small number of these are offered as open source. We want to use an engine that is as algorithmically advanced and graphically realistic as possible. We considered using an open-source clone of the Quake 3 Arena game engine. This code is free, and the in-game realism is acceptable. However, the code is insufficiently documented, making expansion and manipulation of the code difficult. The Torque game engine comes with full documentation and tutorials, so it will be much easier to learn.

3.5 Testing Methods

The testing methods for the decoder and terrain generator are simple: assuming viable input data, check to see if the correct output has been created. In the case of the terrain generator, it is necessary to run the program on the same audio file multiple times to ensure that the same output file is created.

To test the real-time processor, custom OpenGL software will provide for waveform, spectrum, and other visualizations for testing. All prototypes will be written in C/C++ and will run on the SGI workstations in the Engineering West lab. Test data will be song files available at the MARSYAS web site.

Two testing methods will be utilized on the game engine. Modular testing will be done to check each code change as soon as possible. Rigorous integrated testing on the entire system will take place after all of the pieces have been put together. This process will hopefully catch bugs in the software and evaluate the game's performance and resource allocation, with respect to hardware and memory usage.

3.6 Scheduling Diagram with Task Assignments

Objective	October	November	December	January	February	March	April	May
Sound Decoder								
Convert MARSYAS to Windows								
Coding/testing								
Integration								
DTR Generator								
Learn MARSYAS								
Coding/testing								
Integration								
Real-time Processor								
Beat Detection								
Tempo Detection								
Dynamic Detection								
Key Detection								
Graphical Vis. For Testing								
System Integration								
Torque Engine								
Examine existing code								
Modify/enhance code								
System integration								

By the middle of January, we expect to have a digital audio decoder, a terrain generator, a note onset detection system, and a working knowledge of the Torque game engine. When we are successful, we will assemble the sound recognition system and extend the engine to read the DTR file. Once the map can be rendered in the game, we will develop our gameplay additions and user interface. Dan will evaluate the digital audio decoding solution; Jared will implement both music recognition and terrain generation; both Ben and Neil will evaluate our game engine of choice, the Torque game engine.

4. Design Specifications

4.1 Software

The software will be designed using an object oriented methodology. Most of the design will also be done using Microsoft Visual C++ package. We are modifying the Torque game engine for this project so a large number of the classes and functions for creating a game have already been written and compiled together. Our main additions will be the terrain generation and real-time processing.

The terrain generation class will accept an mp3 filename as input. The output will be a complete terrain ready to be sent to the game engine to be displayed. This class can be called before any game execution has been done.

The real-time processing will be done via special classes invoked during game execution. The input will again consist of the name of an mp3 file to be played. The class will play this file as the user plays the game. It will also be doing real time processing. This real time processing will send commands to the game engine to modify the game play, environment, or other factors.

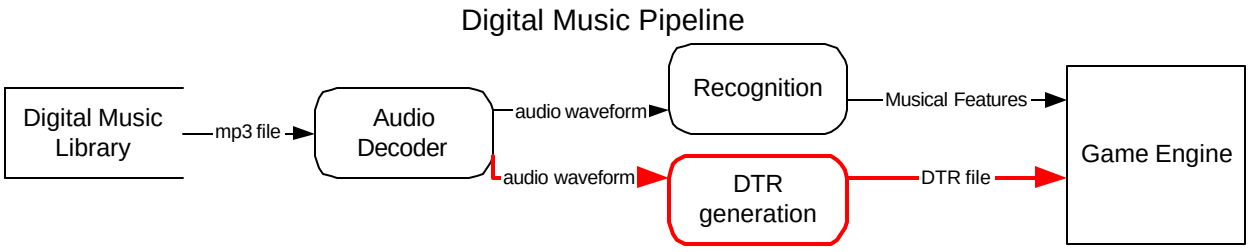


Figure 11: Digital Music Pipeline

4.1.1 Audio Preprocessing

Users will be required to "register" new mp3s before integrating them into game play. This step, which is only performed once per song, is necessary to process and recognize the audio data contained within mp3s before their use within the game. The data produced by this

preprocessing step is used as input into the terrain generator to synthesize a unique terrain for the song (see App. 11.3). For a full discussion of the algorithms employed for recognition, see Appendix 11.2.

The MARSYAS library contains several classes and methods for processing and recognizing audio data. Essentially, these methods iterate over the waveform of an audio signal and apply various filters over different frequency bands to recognize musical features such as tempo, genre, and spectral distribution. The output is written to a file as a "features file," which basically consists of a matrix of numerical values in text form [12].

4.1.2. Terrain Generation

Terrain generation will be performed only once upon the start of a new game. Upon the creation of a new game, the user will be prompted to select a song from the list of registered mp3s to use as input into the terrain generator (see App. 11.3). Using the selected song's corresponding entry contained in the database, the generator will synthesize a terrain resembling an island which uniquely represents the chosen song.

There exist three prominent algorithms which are well-understood and commonly employed to produce synthetic three-dimensional terrain geometry. These are the "fault formation," "midpoint displacement," and "particle deposition" methods [14]. Each method uses a different approach towards producing geometry, each with its own unique visual result (see App. 11.1). The different algorithms themselves take different parameters as input, which can be manipulated to produce predictable results. Depending on the algorithm, we will manipulate the musical features obtained from audio preprocessing into the algorithms' input parameters to produce a terrain appropriate to the song. For example, a measure of a song's "rockiness" will be mapped into a parameter controlling a terrain's coarseness, which would produce a rocky terrain.

4.1.3 Graphical User Interface

The Torque Engine SDK is supplied with an intuitive GUI editor which will make interface development almost trivial. Torque Engine GUIs are designed in an intuitive environment similar to a painting program and can be quickly integrated into a game in development. Different displays for variables such as player health, score, or ammo can be quickly drawn within the editor and tested immediately. These GUIs can even be modified while the game is in play. Finally, there exist several sources online which fully document this process [23], [24], [25], [26].

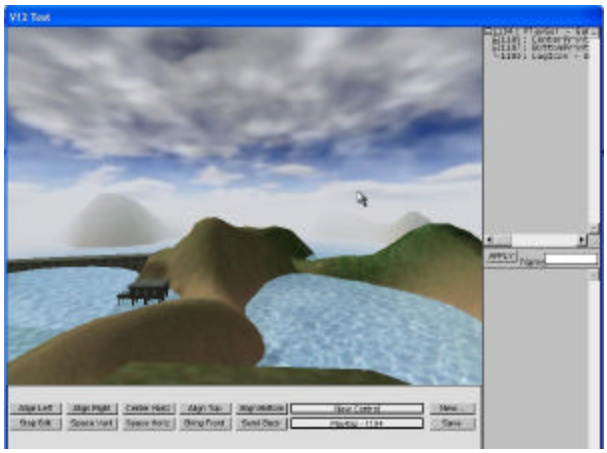


Figure 12: GUI Editor

4.1.4 In-Game Audio Processing

In order to ease the task of integrating in-game processing with the Torque Engine's audio functions, we currently only plan to recognize a few basic musical features in real time. After extensive testing with more complicated algorithms for beat recognition (with lackluster results), we now plan to implement a recognizer for note onsets and dynamics. As opposed to beat detection, which requires complicated measures of musical self-similarity, note onset detection simply requires detecting instances where the overall musical volume quickly jumps above some average threshold value. Dynamics detection simply requires tracking the change of this threshold value.

4.1.5 Game Response to Audio Events

As the in-game audio processing detects onsets and dynamics changes, it will issue events to the Torque Engine's event handler. Specifically, we hope to implement a method

whereby a sudden note onset will affect artificial intelligences by "surprising" or "angering" them. We define "surprise" as causing an AI to suddenly change state into a different behavior. "Anger" is defined as causing an AI to become more aggressive towards the player. In this way, sudden loud notes should catch enemies off guard, perhaps frightening them away, or angering them, causing them to attack the player. A state machine of the AI behavior is shown in Appendix 11.5. It does not include the aforementioned behavior because this behavior will run as a separate process from normal behavior. The Torque Engine of course has already implemented artificial intelligence classes for use within enemies in the game [26]. We also hope to map note onsets into other events such as animations. For instance, while the player stands still, he might tap his foot on every note onset. Particularly salient note onsets may cause lightning strikes, which is another feature of the engine.

Musical dynamics changes will affect the engine in more subtle ways. The drift in volume of a song may affect things such as weather or colors. For instance, we hope for louder songs to produce faster winds and higher ocean waves. A mellow song could produce a clear blue sky, while a heavy song might create a red sky. Again, all these graphical features are supported within the engine [26].

4.1.6 In-Game Events

Like most game engines, the Torque Engine operates within a script-driven environment [26]. These scripts act as a sort of meta-language which reduces defining complex game behavior into writing simplified scripts to control events. Rather than modifying actual game engine source, we hope to simplify the process of game design by writing and modifying currently existing scripts to define the unique properties of our game.

4.2 Data Requirements

Our project's data requirements include both the game environment and behavior data specified by the Torque Engine and the audio and terrain data specified by our modifications.

4.2.1 Terrain Representation

Nearly all three-dimensional terrain data implementations are represented by so-called "height maps" [15], [16]. Our game will be no different. Essentially, a height map is a square grid where each row and column represents a height. In this way, a height map is no different than a square two-dimensional image, where each pixel represents a height instead of a color. Our implementation will store the height map as a two-channel image file. In other words, each point in the map will consist of two values, an 8-bit height, and a flag byte. The flag byte will signal that other objects are to be placed on the terrain at that particular point. For instance, one of the bits will correspond to "tree." If the tree flag is set, a tree is placed at that point. Another bit will correspond to animal. We will used the remaining 6 bits to implement other features as time allows. Finally, a translation module will encode the height data into a format recognizable by the Torque Engine.

In addition to terrain geometry, the terrain generator will create a corresponding mission script to properly describe the island generated. Parameters included in this script will specify water levels, object placement, and climate factors. A full discussion of game scripts follows below.

4.2.2 Audio Representation

The MARSYAS classes represent raw audio data as vectors of floating point numbers. This vector is contained within the Signal class, which also provides methods for piping the data into filters, which are abstracted in the System class. In order to process a "window" of audio

input, the current window of floats is requested from Signal. Next, this vector is sent through a sequence of Systems until the desired output is produced [12].

MARSYAS also provides a rudimentary class for playing audio data; however, our target platform of Windows is not supported. In order to use this class, our team must extend the class for Windows support. Another option is to implement audio by extending the Torque Engine's own audio support, which is based on an OpenAL framework [26].

External to our software, digital audio is encapsulated into a variety of file formats. We have selected the ubiquitous MP3 standard as our format of choice. This standard not only encodes audio, but also compresses its size. In order to extract the raw digital audio data encoded in MP3 files, we will employ the use of openly available decompressor code. Once the audio is in a raw format, it is recognizable by the MARSYAS classes [12].

4.2.3 Game Environment Representation

The in-game environment is described through script files, designated with a .mis (mission) extension. The language in which these scripts are written largely resembles a sort of simplified C code (see App. 11.4) whose primary purpose is to describe the objects and terrain which comprise an instance of a game situation. Objects such as terrain, vegetation, and the sky are instantiated and described with parameters such as position, scale, rotation, and direction. Upon game initiation, the proper script is interpreted, which instantiates the objects and parameters specified [23], [24], [25], [26].

4.2.4 Flow of Control Representation

The in-game flow of control is also described through script files, designated with a .cs (control script) extension. These files describe function calls and flow of control much like a familiar C or Perl program [23],[24],[25],[26].

4.3 Hardware

Our team's project requires no special hardware beyond the constraints of the target platform. To run it, the end user must supply a computer platform that meets or exceeds the requirements presented given by the Torque Game Engine. These requirements are a Win98/2000 platform with a 400 Mhz Pentium II PC with 64 megs RAM [21].

5. Design Solution

5.1. 3D Game Engine

The Torque 3D game engine, which was used to develop the game Tribes 2 by Sierra, was chosen to be the foundation of our game. Source code was obtained from GarageGames.com for $100. This game engine was chosen over open source clones of the Quake 3 engine because of the massive on-line community for the Torque engine. We have made extensive modifications to the game engine source code and its scripts in order to develop our game.

5.2. Audio Decoder & Recognizer

A tool called MARSYAS was used to decode and recognize aspects of audio files. The data received from this is used to generate a unique terrain and mission for each song.

5.3. In-Game Audio

OpenAL, which was built into the Torque Engine, is used for sound effects.

Since OpenAL does not currently support mp3 audio, Bass was used to accomplish in-game mp3 playing as background music.

5.4. Installer

Since we were building a game, be realized that we would have to create an installer so that people could easily install the game. Since gaining a license for InstallShield is very expensive, an open source clone, called Inno Setup v2.0.19, was used.

6. System Implementation

7. System Performance, Testing & Evaluation

"Cannibal Island" runs very well. While we do not actually have a computer with the same specs as our previously stated target machine, the game does run very well on high end machines. Our test machines are as follows:

- 2 x 1.4 GHz Processor w/ 512 MB RAM
- 2 x 1.2 GHz Processor w/ 512 MB RAM
- Dual 1 GHz Processors w/ 512 MB RAM
- 300 MHz Processor w/ 160 MB RAM

While the 1.4 GHz machines achieved frame rates of about 50 per second, the 300 MHz machine only achieved about 6. After some testing, we determined that this low frame rate on the 300 MHz machine was mostly due to having computer controlled opponents in the game. When they were removed, the frame rate increased to about 10. Mp3 playing has negligible effect on game performance.

8. Conclusions

The "Cannibal Island" project consists of four main segments: the audio decoder, terrain generation, real-time processing, and the game engine. Each segment contains its own unique challenges, all of which are reachable. The audio decoding will be accomplished by analyzing and implementing open source algorithms currently utilized by digital music players, such as FreeAmp. By analyzing and extending MARSYAS algorithm and code, certain properties of the song can be obtained. Those properties will be used to generate unique terrain. Real-time processing of the song using MARSYAS will allow for properties to be extracted for in-game effects. The Torque Engine, with modifications, will be more

than enough to suit the 3-D aspirations of this game, as detailed above. Once all segments of this project are working together, the user will be able to customize a game like never before, with music.

With the development of the audio decoder, the terrain generator, and modifications to the graphics engine, "Cannibal Island" will be a success. A new era of video games will evolve. The incorporation of user-provided music will enhance the replayability of the game. With each new song, a new island will be created. The environment will behave differently; lighting will strike, rain will fall, or the sun will shine brightly. Computer-controlled agents will behave in a different way each time. A new story will be told.

9. Future Work

Since we only have a very basic artificial intelligence in the game, this area could be greatly expanded upon. The current behavior of the "bots" is described in Appendix 11.5. In the future, we could make the bots behave differently depending on what song is currently being played. They could run faster or slower, be more aggressive or passive, to list a few possibilities.

Also, our game lacks what makes most games very successful: a storyline. Currently in our game, there is only terrain generation and game play, there is no back-story answering why the player is on the island or real goal to achieve in the game. Our original hopes were to get the player to have to find a flare gun and then shoot a flare when a rescue plane came close enough to the player, ending the game. Do to time constraints and lack of knowledge of how vehicles worked in the game engine, this has not been implemented yet.

10. References

1. Allen, Paul E. (**1999**). "Tracking Musical Beats in Real Time," School of Computer Science, Carnegie Mellon University.

2. Cummins, Fred. *Bex: Documentation for a Beat Extractor*. 26 Sept. 2001 <http://www.cs.indiana.edu/~port/bex/Bex.html>.
3. Dixon, Simon. (1999). "A Beat Tracking System for Audio Signals." Austrian Research Institute for Artificial Intelligence.
4. Dixon, Simon. (2001). "Automatic Extraction of Tempo and Beat from Expressive Performances." Austrian Research Institute for Artificial Intelligence.
5. Foote, Johnathan (2001). "The Beat Spectrum: a New Approach to Rhythm Analysis." IEEE International Conference on Multimedia & Expo 2001, Electronic Procedings.
6. Goto, Masataka et al. (1998). "An Audio-Based Real-time Beat Tracking System and Its Applications." School of Science and Engineering, Waseda University.
7. Goto, Masataka et al. (1998). "Issues in Evaluating Beat Tracking." School of Science and Engineering, Waseda University.
8. Large, Edward W. et al. (1999). "Resonance and the Perception of Musical Meter." Connection Science, 6 (1), 177-208.
9. Scheirer, Eric D. (1998). "Tempo and beat analysis of acoustic musical signals," Acoustical Society of America 103, 588-601.
10. Scheier, Eric D. "Beat-tracking algorithm." (from subject-line) Online posting. 3 May 1999. Available: music-dsp@shoko.calarts.edu. 26 Sept. 2001. Available: http://shoko.calarts.edu/musicdsp
11. Slotau, Hagen et al. (1999). "Recognition of Music Types." Interactive Systems Laboratories, University of Karlsruhe (Germany), Carnegie Mellon University (USA).
12. Tzanetakis, George, Essl, Georg, Cook, Perry (2000). "Audio Analysis using the Discrete Wavelet Transform," Princeton Computer Science Department.

13. *Writing Plug-ins*. Nullsoft Developer Network. 30 Sept. 2001. Available: http://www.winamp.com/nsdn/winamp2x/dev/plugins/.

14. DeLoura, Mark. *Game Programming Gems.* Charles River Media, 2000.

15. *Terrain Tutorial*. Lighthouse 3D. 30 Sept. 2001. Available: http://www.lighthouse3d.com/opengl/terrain/.

16. *Welcome to the Virtual Terrain Project*. Virtual Terrain Project. 30 Sept. 2001. Available: http://www.vterrain.org.

17. *Torque Games Engine/GarageGames FAQ.* Garagegames.com. 8 Oct. 2001. Available: http://www.garagegames.com/index.php?sec=mg&mod=v12&page=faq.

18. *Super Mario 64*. Nintendo.com. 8 Oct. 2001. Available: http://www.nintendo.com/games/gamepage/gamepage_main.jsp?game_id=249.

19. *Writing your first render function*. The Developers Gallery. 30 Sept. 2001. Available: http://www.dev-gallery.com/main_page.htm

20. Hipp, Michael. *Mpg123, Fast MP3 Player for Linux and UNIX Systems.* Mpg123 Project. 9 Oct. 2001. Available: http://www.mpg123.de.

21. *Sierra: Tribes 2 – Team Combat on an Epic Scale*. Sierra. 9 Oct. 2001. Available: http://tribes2.sierra.com/.

22. *Fees for Fall 2001*. MU Enrollment Services. 9 Oct. 2001. Available: http://www.missouri.edu/~regwww/MU_Registration/registration/fees_fall.shtml.

23. *V12 Documentation Repository*. V12 Ultima Group. 7 Dec. 2001. Available: http://v12.ultimagroup.com.

24. *V12 Powered*. Gameznet. 9 Oct. 2001. Available: http://gameznet.com/v12/.

25. *V12 Mod Central*. MGO.NETwork. 7 Dec. 2001. Available: http://v12.mgonetwork.com.

26. *Torque Engine Documentation.* Garagegames.com. 7 Dec. 2001. Available: http://www.garagegames.com/index.php?sec=mg&mod=v12sdk&page=doco.

11. Appendices

11.1. Terrain Generation

11.1.1. Fault Formation Algorithm:

```
generate_terrain_fault_formation(minHeight, maxHeight, n)
{
    1.  Initialize a height map to minHeight
    2.  for i = 1 to n
            a.  Draw a random line across the height field
            b.  Calculate dHeight_i
            c.  Add dHeight_i to each value on one side of the
                line
}
```

Where:

$dHeight_i = minHeight + (i/n)(maxHeight - minHeight)$

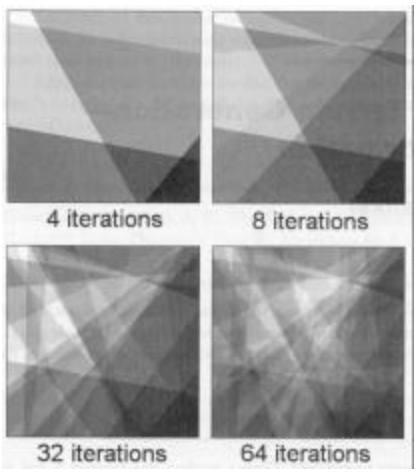


Figure 13: Fault Formation Examples

This algorithm synthesizes terrain by generating random "faults" across a height map and shifting all the "land" on one side of the fault by an offset of dHeight, which is a function of i, and the height boundaries. Larger values of n will produce more iterations and a greater level of detail.

11.1.2. Midpoint Displacement Algorithm:

```
generate_terrain_midpoint_displacement(r,dHeight
,square)
{
       1.  if dHeight < threshold stop recursion
       2.  else
              a.  calculate the height at the
                  midpoint of the square as
                  midpoint = average of four
                  corners + random(-
                  dHeight/2,+dHeight/2)
              b.  dHeight = dHeight*2^-r
              c.  generate_terrain_midpoint_dis
                  placement(r, dHeight,
                  bottomRightSquare)
              d.  generate_terrain_midpoint_displacement(r, dHeight, bottomLeftSquare)
              e.  generate_terrain_midpoint_displacement(r, dHeight, topRightSquare)
              f.  generate_terrain_midpoint_displacement(r, dHeight, topLeftSquare)
}
```

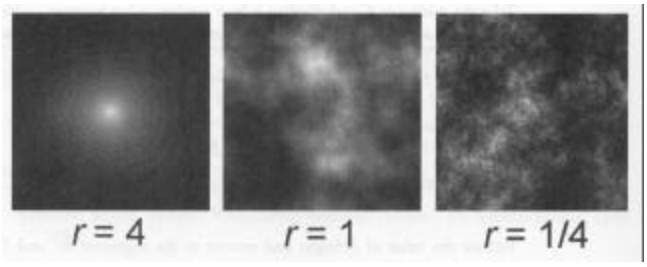


Figure 14: Midpoint Displacement Examples

This algorithm takes as input a map with a randomly chosen initial condition at each corner. The first iteration will input *dHeight* as the length of the diagonal across the initial height map. Next, if *dHeight* is larger than some pre-specified threshold value, the midpoint of the square is set to the average of the four corners plus a random value in the range (-*dHeight*/2, *dHeight*/2). *dHeight* is updated by multiplying it by 2^{-r}. The recursion continues on the four new squares created by dividing the initial square at the midpoint.

11.1.3. Particle Deposition Algorithm

```
generate_terrain_particle_deposition(n)
{
       1.  Initialize a height map to zero
       2.  for i from 1 to n
              a.  location = random(height map)
                     i.  location += dHeight
                    ii.  while at least one of
                         location's neighbors is at a
                         lower height
                             1.  location -= dHeight
                             2.  location = random(one of
                                 location's neighbors)
}
```

This algorithm simulates the buildup and cooling of lava on a flat surface. The algorithm drops particles at a random location on the height map and agitates them until they come to rest (step 2.a.ii).

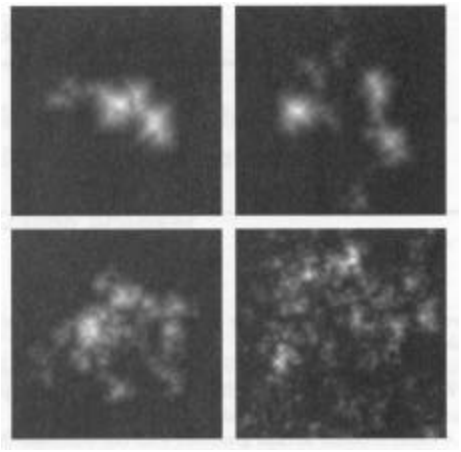


Figure 15: Particle Deposition Examples

11.1.4. Algorithm Render Examples

Figure 16: Fault Line Algorithm Render

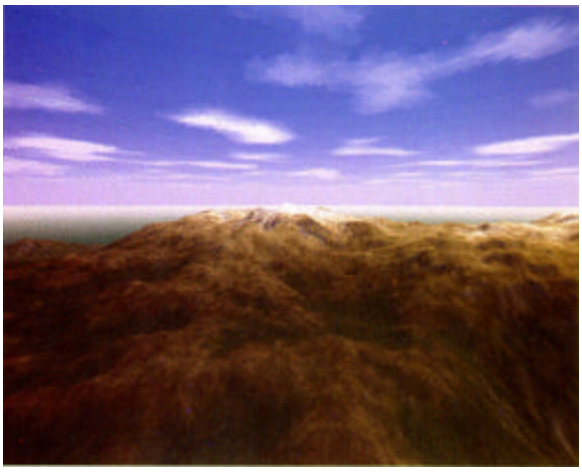


Figure 17: Midpoint Displacement Algorithm Render

Figure 18: Particle Deposition Algorithm Render

11.2. Beat/Tempo Recognition Algorithms

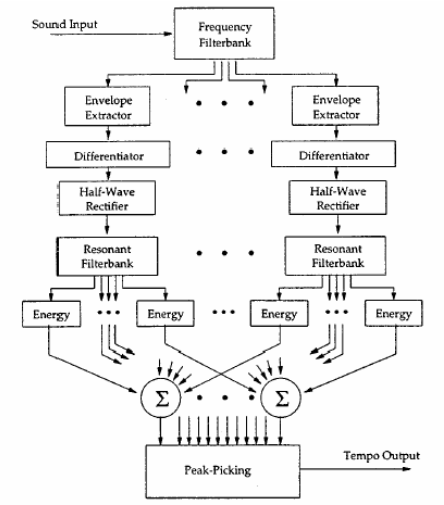
11.2.1. Simple Note Onset Detection

```
detect_onset(current_audio_window, last_FFT, threshold)
{
       1.  current_FFT = fast_fourier_transform(current_audio_window)
       2.  compare current_FFT against last_FFT
           2.1 for each subband in FFTs
               2.1.1   if sum(current_subband[i]) – sum(last_subband[i]) > threshold
                       2.1.1.2 register_note_onset(whichSubband)
}
```

This simple algorithm simply sums the energy in each frequency subband of the current audio window and compares them to the corresponding subband in the last window's FFT. If the difference in energy is greater than some threshold, then we register an onset. The choice of a threshold is fairly arbitrary, but should be proportional to the overall audio level of the song. The choice of subband division is also fairly arbitrary, depending on which frequencies in which we are most interested. For example, most of the energy of a pop song will be concentrated towards lower frequencies. Spikes in sum differences will correspond to bass beats.

11.2.2. Tempo Detection – Schierer Algorithm

```
detect_tempo_schierer(current_audio_window)
{
        1.  current_FFT =
            fast_fourier_transform(current_audio_window)
        2.  divide current_FFT into frequency_subband[1..S]
        3.  for each frequency_subband[1 .. B]
                a.  inverse_FFT =
                    inverse_fast_fourier_transform(frequency_s
                    ubband[b])
                b.  envelope = low_pass_filter(inverse_FFT)
                c.  derivative =
                    first_order_differentiator(envelope)
                d.  rectified =
                    full_wave_rectifier(derivative)
                e.  for each comb_filter[1 .. N]
                        i.  energy = comb_filter[n](rectified)
                       ii.  total_energy[n] += energy
        4.  phase = peak_picker(total_energy[1 .. N])
}
```



19: Scheirer System Diagram

This rather complicated algorithm estimates the tempo (or inverse phase) of an audio waveform by applying signal processing techniques. First, the input is divided into frequency subbands, and their envelopes are extracted. Next, a differentiator finds the first order difference function to approximate the derivative. The derivative is full-wave rectified to double the envelope's accuracy.

The second phase of the system is to send the signal through a resonant filterbank, comprised of N comb filters. Each filter is associated with a unique delay, which represents a phase, or equivalently, a tempo. Essentially, at this step, the algorithm attempts to match the signal with a corresponding tempo somewhere in the filterbank. In other words, in order to recognize a specific tempo, there must exist a resonator within the filter bank with the corresponding delay, or the wrong tempo will be estimated. This means the algorithm's precision is proportional to N, the number of comb filters.

Finally, a peak-picker finds the maximum value within the *total_energy* vector, and the resulting tempo is estimated as the corresponding comb filter's delay.

Further calculations can be performed to estimate when the next beat will fall in the future.

Obviously, this algorithm is extremely expensive. While Schierer notes that the number B of initial frequency channels and band divisions is somewhat arbitrary (B ~ 5), for accuracy, the number N of filters must be on the order of 150.

With the numbers Schierer quotes for accuracy, we find that in order to produce acceptable results, an audio waveform must pass through nearly 750 systems 150 times a second! Considering both the complexity of the implementation and computations of this algorithm, our team may attempt an implementation only if time and resources allows.

11.2.3. Tempo Detection – Tzanetakis, et al. Algorithm

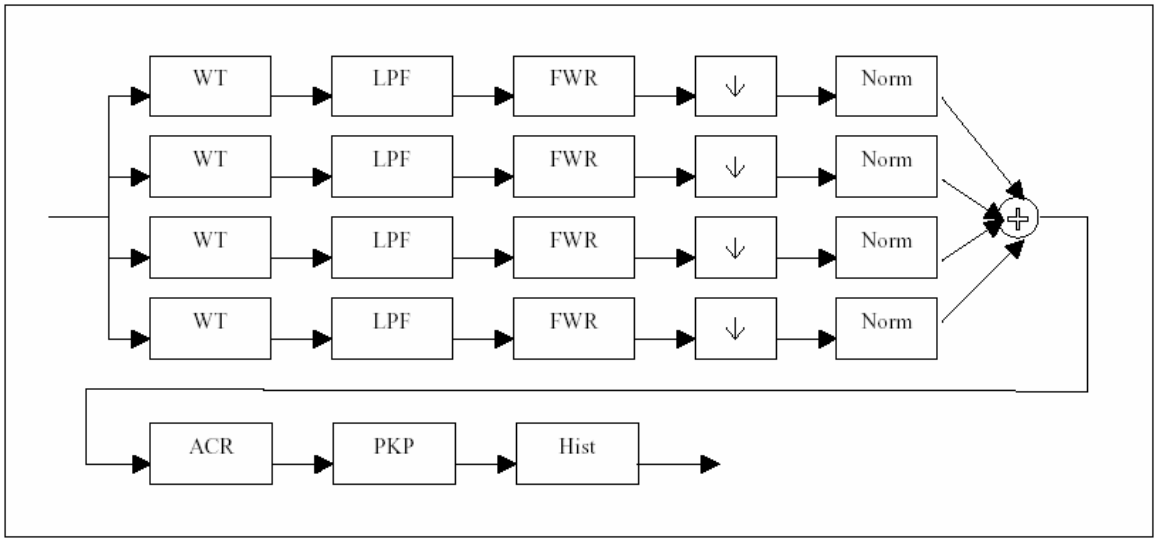


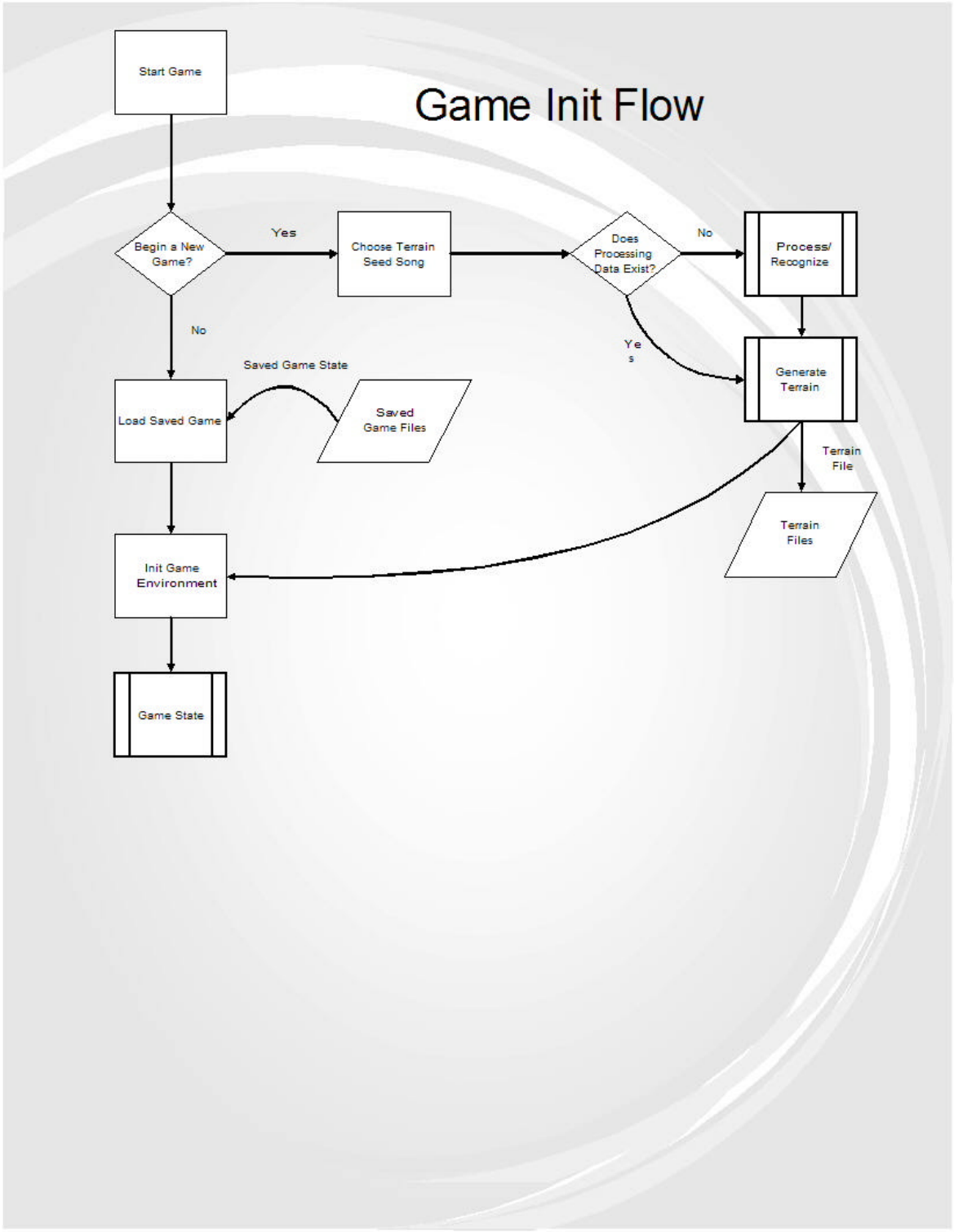
Figure 20: Tzanetakis System Diagram

```
detect_tempo_tzanetakis(current_audio_waveform)
{
        1.  current_DWT = discrete_wavelet_transform(current_audio_waveform)
        2.  divide current_DWT into frequency_subband[1 .. S]
        3.  for each frequency_subband[1 .. S]
                a.  inverse_DWT = inverse_discrete_wavelet_transform(frequency_subband[b])
                b.  envelope = low_pass_filter(frequency_subband[b])
                c.  rectified = full_wave_rectifier(envelope)
                d.  downsampled = downsampler(rectified)
                e.  norm = normalize(downsampled)
                f.  envelope += norm
        4.  autocorellated = autocorrelator(envelope)
        5.  tempo = peak_picker(autocorrelated)
}
```

This algorithm is similar in some ways to Schierer's. Recognition is performed across frequency subbands. Like Schierer, integration of frequency bands is left until the end. Rather than FFT, the algorithm employs the Discrete Wavelet Transform for frequency division, but this difference is rather minor.

The key difference in this algorithm is the recognition step, which is performed in one call to an autocorrelation system. Unlike Schierer's algorithm which searches for a tempo across a resonator frequency bank, Tzanetakis performs a simple peak search across an autocorrelation function. However, since the autocorrelation system discards phase information, there is no way to predict when the next beat will fall. Therefore, this algorithm simply calculates beats per second (or frequency), where Schierer's calculates both tempo and beat onsets (phase).

11.3. Game Init Flowchart



11.4. Torque Engine Sample Script

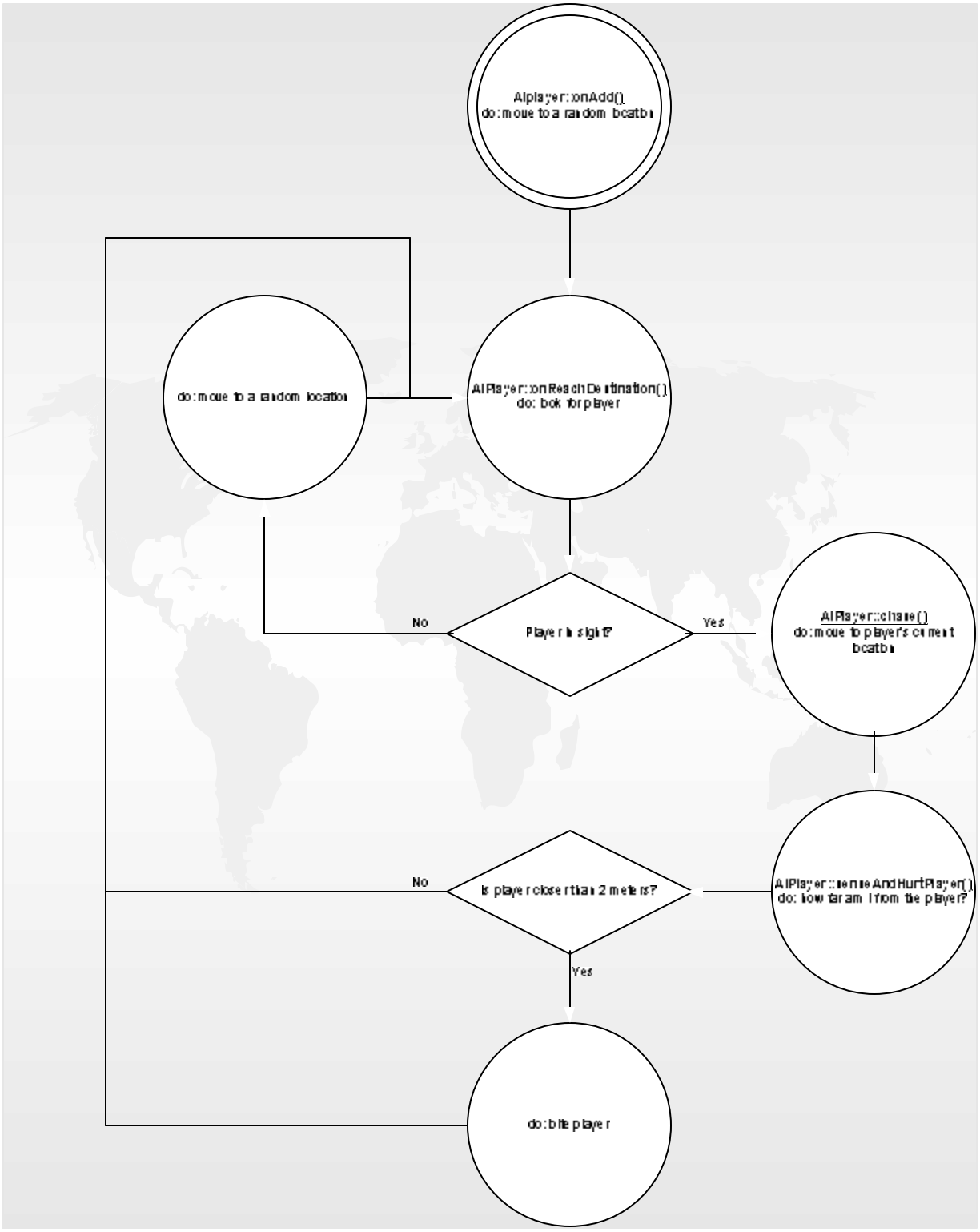
```
//--- OBJECT WRITE BEGIN ---
new SimGroup(MissionGroup) {

   new ScriptObject(MissionInfo) {
         desc1 = "The V12 Engine SDK, Test Mission.";
         name = "Test (quick load)";
         desc0 = "A very simple test mission with minimal textures that loads very quickly.";
   };
   new MissionArea(MissionArea) {
      area = "-360 -648 720 1296";
      flightCeiling = "300";
      flightCeilingRange = "20";
         locked = "true";
   };
   new Sky(Sky) {
      position = "336 136 0";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      cloudHeightPer[0] = "0.349971";
      cloudHeightPer[1] = "0.25";
      cloudHeightPer[2] = "0.199973";
      cloudSpeed1 = "0.0001";
      cloudSpeed2 = "0.0002";
      cloudSpeed3 = "0.0003";
      visibleDistance = "500";
      useSkyTextures = "1";
      renderBottomTexture = "0";
      SkySolidColor = "0.547000 0.641000 0.789000 0.000000";
      fogDistance = "300";
      fogColor = "0.820000 0.828000 0.844000 1.000000";
```

Figure 21: Script Snippet

This script snippet is an example of the definition of a mission for the Torque Engine. This object-oriented approach to game design simplifies defining game environment and behavior.

11.5 Artificial Intelligence



The artificial intelligence for “Cannibal Island” is fairly simple. Once added to the game, the computer controlled opponent computes a random location on the map and then moves to it. Once it gets to that location, it looks for a human controlled player. If there is one within a radius of 75 meters, it moves to the point that human player is currently located. Once it gets there, it will look for the human player again. If that player is now within 2 meters, the computer controlled player will bite the human player, inflicting damage. However, if the computer controlled player cannot see a human player, i.e. the closest human player is farther than 75 meters away, it will compute another random location and then move to it.

The AI in the purchased engine was very sparse. It only contained functions which helped control movement. All the functions described above were written in the scripts and utilized the movement functions that came with the engine.