April 2009

iGotBand: Improvisational Music Game

Brian Takumi-Miyaji Hettrick
Worcester Polytechnic Institute

Michelle L. Clifford
Worcester Polytechnic Institute

Timothy Barrett Cushman
Worcester Polytechnic Institute

Worcester Polytechnic Institute

Improvisational Music Game

Major Qualifying Project Report

Model	PRS Custom 22
Frets	22
Fretboard	Rosewood
Neck	Mahogany, Wide Fat
Scale Length	25"
Body	Maple Top, Mahogany Back
Pickups	Dragon II Treble and Bass

Modeled and Textured by Brian Hettrick

Michelle Clifford Timothy Cushman Brian Hettrick

4/30/2009

Table of Contents

Table of Images

1. Introduction

1.1. Acknowledgements

The first acknowledgement we would like to make is to our advisor, Joshua Rosenstock, who kept us on track when our wheels of creativity would spin off axis.

Secondly we would like to acknowledge the assistance of Alex Schwartz who assisted us in pre-production as well as the modeling and animation of the characters. He also took the time to code a MEL script to systematically and identically render each of the characters. In addition, he created the interface for the game menu layout.

1.2. Preface

iGotBand is a Flash based guitar game in which players are confronted with oncoming fans in three different types of venues. Each fan has over his head a set of notes that if played give them the incentive to continue to listen to the player play. The game is constructed such that it allows players to play whatever they want because at no point does the game actually require the player to play at a certain level of perfection to proceed.

1.3. Existing Music Games

iGotBand is an effort to diverge from the state of music games as they are today in an effort to promote improvisation and interpretation of the music that the player is presented with. For this reason the players at no point during the game are required to fulfill any requirements of quantified achievement within the song to proceed further. Since the game is focused on the ability of players to improvise the songs that they are given, it would be nearly impossible to judge players on their improvisational skills with any kind of traditional numeric system of scoring. In absence of a scoring system that uses numbers to gauge the player's skill, the game instead uses the somewhat abstract system in which fans are collected by players if the player is able to follow the guide of notes that is presented to him.

Figure 1.3:1 Concept of iGotBand logo.

1.4 Game Design Document

GAME
Language - Actionscript [Adobe Flash]
Players - 2 player
Input Device - USB Keyboard OR USB Compatible Instrument peripheral

GAMEPLAY
iGotBand will be played in a Flash document, on our website or downloaded to the computer. The player will play as a guitarist in a virtual band, and will collect fans by playing the correct notes indicated by each fan.

NPCs
NPCs are fans that approach the players by walking along a set path in the level. Fans will have specific notes that they want to hear in order to be "captured" and brought to the listening area near the band stage. Fans come in different styles, such as the goth girl and the punk rocker, but all fans will appreciate their requested notes being played no matter their style.

LEVELS
Each level is a different playing venue. In each level there is a designated area for the band to play (the stage), an area with a path set for fans to walk by the stage, and an area near the stage where "captured" fans listen intently to the music and cheer on the band.

Venues include the City Park, the Bar, and the Stadium.

IMPROVISATION
iGotBand is an effort to stray away from the current state of music games as they are today, in an effort to promote improvisation and interpretation of the music that the player is presented with. For this reason the players at no point during the game are required to fulfill any requirements presented within the song to proceed further. As mentioned under NPCs, the fans have over their heads a set of notes that if played would give them the incentive to continue to listen to the player play. To "capture" a fan and have it move near the listening area, the player must play these notes exactly as seen. There is no penalty for not playing these notes at all and not capturing the fan, nor is there a penalty for capturing fans out of order, or playing extra notes in between capturing two fans.

2. Foundation of the Process

2.1. Improvisation

Throughout the beginning of the project, we spent a considerable amount of time looking at different forms of improvisation in preparation for the design of the game. Although many of the ideas that we reviewed would not be possible because they were not as much musically related as they were forms in which improvisation could take. The study of these forms of improvisation however were a good first step in order to formulate a concept as to what form the final project would take in terms of game play.

2.2. Opportunities of the Internet

It was established very early in the development process that the game would be based on the internet in some way shape or form. This was decided upon not only because the many benefits that this affords us as developers and the players, such as ease of distribution and access, but because internet based games are the future of all gaming. This can be seen with the move on the part of consoles to become greatly involved in internet game play, as well as most games now having the ability to be paid for and downloaded over the internet.

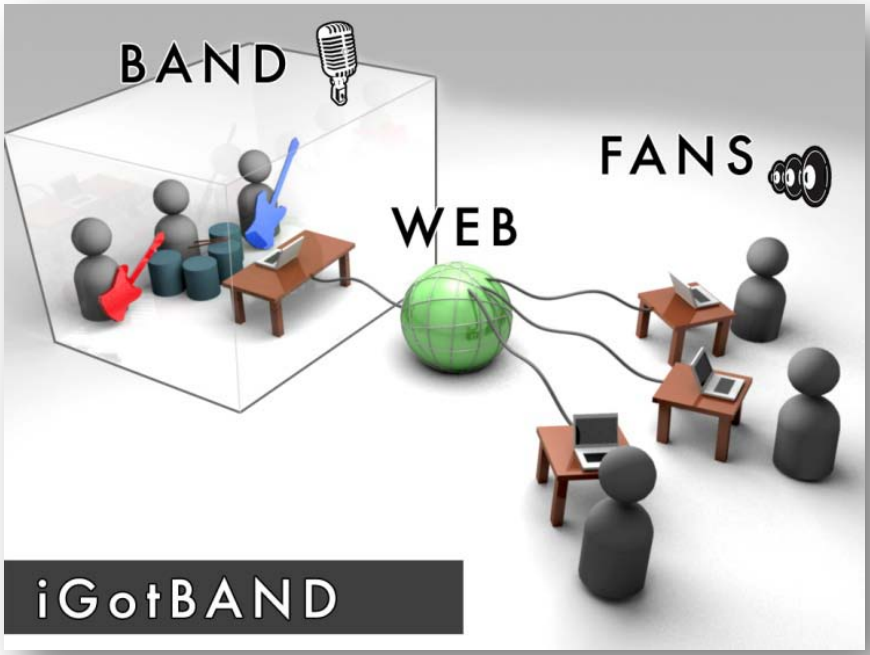


Figure 2.2:1 Concept for iGotBand online play.

2.2.1. *Picking from Endless Possibilities*

With the great number of opportunities that were now at our fingertips while developing the game on the internet, the breadth of the game started to. The development quickly turned from what we can do with the game, to how much stuff we can put in it. This tangent led us away from the traditional game style that we had first envisioned, but was worthwhile in the end because of the ideas that were a result, moving the game away from the early concepts that had been produced and into something that had tangible assets that would could develop into a more cohesive and orderly game.

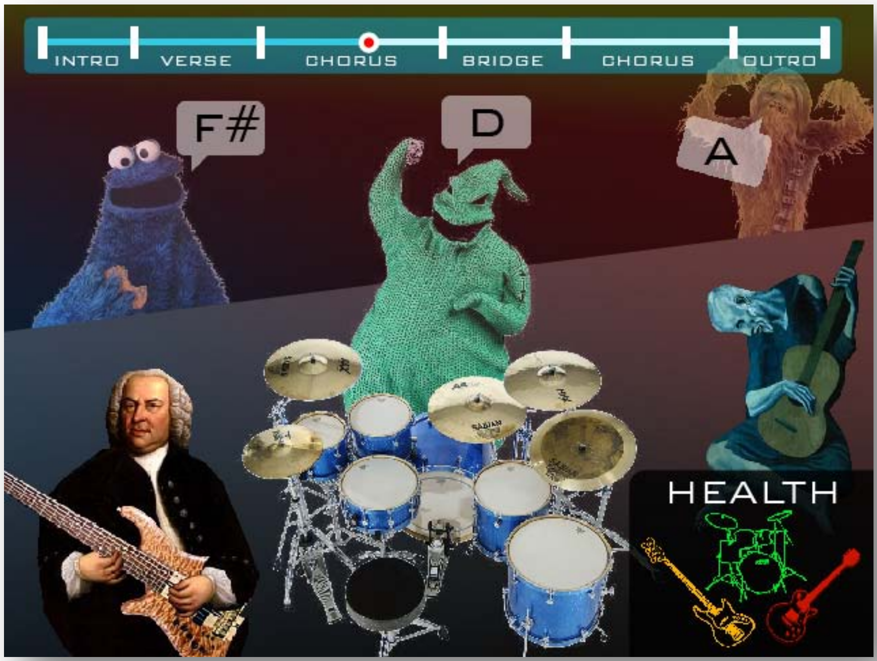


Figure 2.2:2 Early concept of gameplay

2.2.2. *Ease of Access*

Accessibility to the game was another element that was considered early on. In addition to being distributed on the internet it was decided upon by the group that the game would be engineered in Flash to increase the available player base because of Flash's ability to play on almost any computer. Another benefit to Flash is that it can be played without actually having to download the game, so it can therefore be played on any computer that does not allow the user to install programs, but has Flash Player previously installed.

3. Development of the Design

3.1. Chords of the Dead

Our first design idea was a game where players would have to fend off unruly concert-going fans by playing the correct notes in addition to whatever else they were playing. This design introduced the abstract ideas that we wanted to keep for future design iterations.

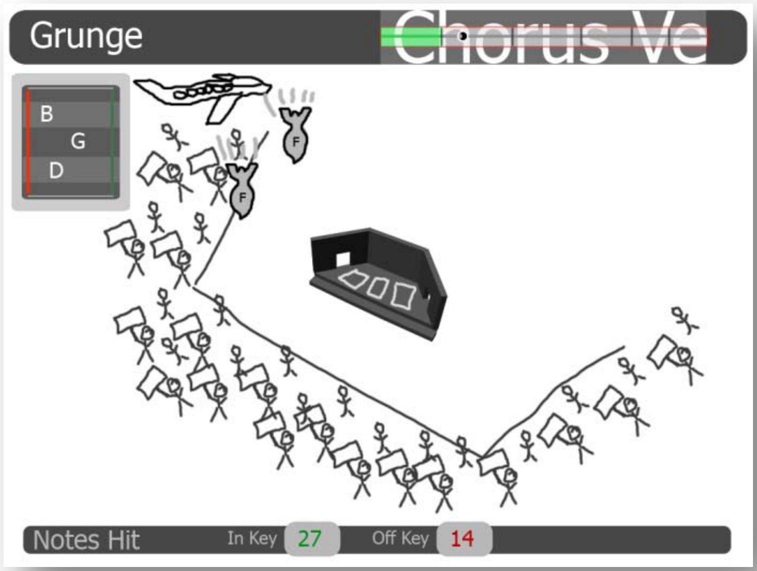


Figure 3.1:1 Early concept of gameplay.

3.1.1. *Focus on the Fans*

As a game that centers on improvisation, we did not want to score players strictly for hitting notes. We also, however, did not want to leave the game too open-ended. In this design iteration, we developed the concept of presenting notes to hit and also a scoring mechanism based on the notion of fans.

3.1.2. *Multiplayer*

This original game concept also introduced the core element of multiplayer that we wanted to keep in future design iterations. Our idea of a multiplayer band consisted of a guitar, a bass, and drums. Having a strong emphasis on multiplayer play would increase the re-playability of our game.

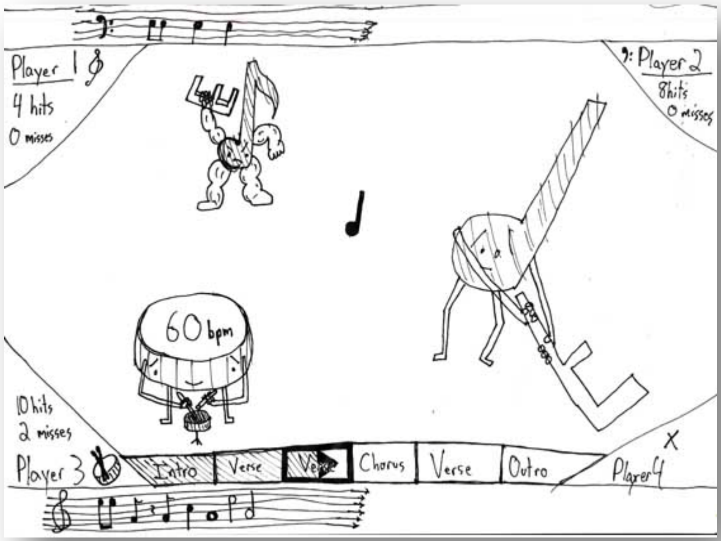


Figure 3.1:2 Early concept of gameplay

3.2.ImprovOnline

3.2.1. Community Based Play

ImprovOnline's core design is player-driven content. Bands or artists of other talent would stream their work online through a designed website, where others would be listening and giving feedback in real time. Artists and listeners would accrue points to be able to obtain trophies or items brandishing band logos. This type of game play would be very much focused in social interaction as a result.

3.2.2. The Cons of ImprovOnline

Many issues with this idea cropped up through discussion. First and foremost, this game design would require an audio input, like a microphone. The average computer microphone might not give recorded or streamed content a desired sound quality, so this online Improv scenario calls for potential bands to have more advanced microphone peripherals. This is not something we would like to rely on. Also, the game would have to be designed with internet features in mind. Much website coding would be involved, and then a lot of thought would need to be put into how the website would store all of the user data and content. We would need to implement a login system and be able to manage however many users would be created over time. There would be a lot of possible copyright issues involved. How would artists be able to host their work without fear of it being stolen? What if the site hosts covers to already existing songs? We cannot rely on the users to be completely honest.

The community would require moderation. A setting where every player involved has free reign could easily give way to acts of grief from community members who have nothing better to do. Community members with bad attitude may especially run rampant when armed with the knowledge that not all of the content posted would be amazing; more than likely the site would get a lot of mediocre songs. Over time, a steady player base is needed. The type of design this game implements is not wholly unique, as bands already utilize websites like MySpace and Youtube to post their content. The fact that these sites exist might not deter a player base from forming at all, but if bands have already formed a hub online where they stream their shows, they might not feel compelled to move to this website. As the content is player-driven, if there are no players, there is basically an empty shell of a game with nothing going on in it. If there were no bands hosting their content, no one is going to want to walk around with nothing to listen to and give feedback on.

3.2.3. "The Road"

A big concept we thought of during this design iteration was the idea of "The Road", a linear path of progression that would be present in the game. A path like this would make the intended progression obvious to the player and reinforce achievement positively. This concept lived through ImprovOnline and found its way into our final design iteration.

3.3. iGotBand Flash

3.3.1. Pulling Back

This last iteration had us moving back to the single computer and away from the big online picture. Our original abstract ideas of the use of fans to offer notes to the players and act as a scoring mechanism made their return. Improv would still be a key factor in game play. There would be no penalty for deviating from the given notes, nor is there one for not playing the given notes at all. A structure where players attract fans by playing the notes that they offer gives a base for experts to play off of and provides a musical backbone for beginners.

3.3.2. A Game that is Compelling and Concise

Our concept of the "The Road" also made its return here. Literally, the road is where fans will walk, and you catch their attention by playing correct notes. The idea behind this is that as the fans are walking by they like what they are hearing. This game play is simple to understand, but also is very open-ended, allowing for a re-playability factor.

This idea presents much less clutter than the online community design would have had; there is no longer a need to collect items, so now the game is about playing music, and not playing music to acquire items. It is also more improv-oriented than the online model. Music played within the online model would have most likely been confined to music that was thought to be popular. Players in a massively multiplayer setting would be much less likely to practice improvisation for fear that they would be ridiculed for sounding bad.

Figure 3.3:1 Artistic render of speakers from Arena level

This game would be written in Flash, so it will play on most Mac/PC machines. The game would feature little menu navigation, which would bring the player to the game faster, and keep beginners from getting overwhelmed with having too many options. If a player so chose, they could play the game using a computer keyboard, which not only is something that people with computers generally know how to use, but also is an accessory that comes standard with most, if not all, computers and laptops. It would be downloadable from a web site, and after that the game could be played locally from a computer without internet access. These factors present an ease of access to the game that makes it much easier and more fun to play.

4. Producing a Game

4.1.Tool Overview

Adobe Photoshop CS4 Extended

Adobe Photoshop is a two dimensional graphics editing program that is currently very entrenched as the industry standard for almost every industry that requires graphics processing. Photoshop was used to make all of the website pages as well as touch up the renders of the levels that were first modeled in three dimensions.

AutoDesk Maya 2008

AutoDesk Maya 2008 is a three dimensional graphics editor and modeling program. Maya was used to model and animate all of the levels, characters, and website assets in the project.

Adobe Flash CS4

Adobe Flash was the program used to run the game using ActionScript 2.0 as the coding language.

4.2.Levels

The levels in game were designed, modeled, and rendered by Tim Cushman. All of the three levels offered in the final product were both designed and modeled in Maya 2008, with some tweaks applied to the images rendered out in Photoshop CS4. In all cases, the renders from the 3d modeling program were rendered out using Mental Ray rendering process in Maya 2008, using an isometric viewpoint to produce a single frame. The look of the levels was made such that they appeared simply constructed, and textured in a way to induce the feeling of a sleek cartoon look. Following the design of the light and straightforward user interface, the three dimensional aspects of the levels were made such that were relatively uncluttered, without being too bleak and being seen as desolate. This same approach was used when considering the choices of colors and textures that would be applied to the levels. In many cases, segments within the level that would conventionally be a single piece of three dimensional geometry, with a two dimensional texture and bump map applied, were instead constructed in the 3d environment and textured with simple colors. This was done to ensure the motif of a simplistic look would be existent ubiquitously throughout the game. The simple look was used to give the feel that the level itself was a stage for the game as a whole and a stage for the almost marionette looking character, as opposed to the literal stage showing in the level.

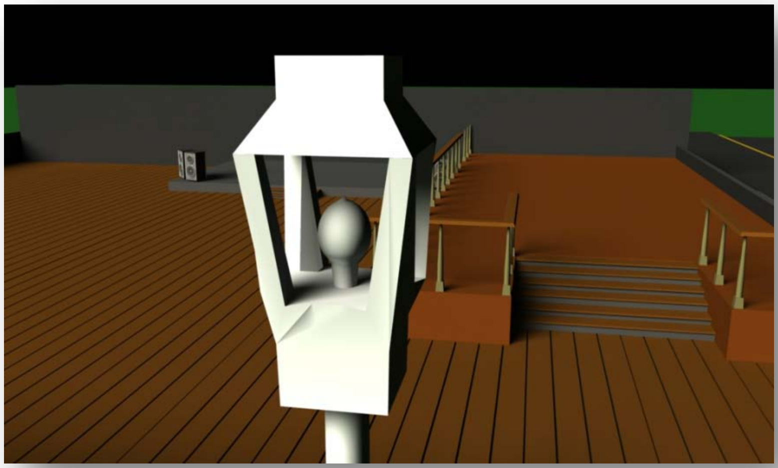


Figure 4.2:1 Artistic render made during Bar level development

There were a few items in the three levels that were constructed or textured without simple geometry and/or texture. In all cases this was done because those items were plain to such an extent that they looked less like a visual interpretation of the object, and more like an assembly of the geometry from which they are constructed. In early stages of the construction of the park level this was made very clear with the trees in the case of geometry that was too basic, and the grass in the case of a texture that was too simple.

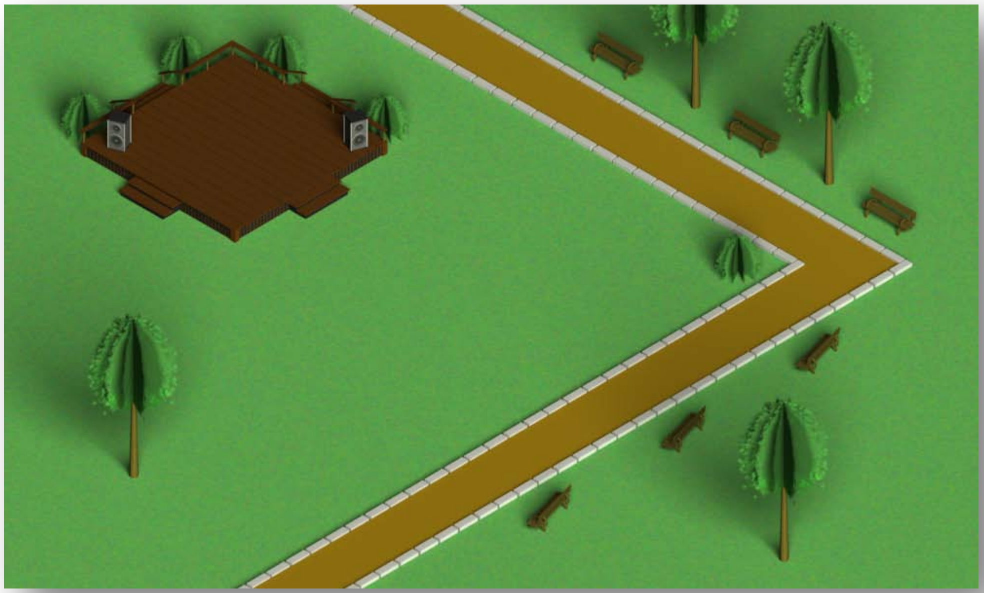


Figure 4.2:2 Early render of Park level

The trees were originally constructed of a central cylinder with planes evenly rotated around its top end. Although this is the conventional process in many games for the construction of trees, in this case since the level in its final presentation is only two dimensional, this way of construction did not give the illusion of "tree" because the player is not given the opportunity to move around it in a three dimensional environment. Another reason that the geometry may not have been synonymous with being a tree is that in most cases in games trees are viewed from underneath, so the aerial view of the same object would seem alien. The trees in the park were instead created by using a paint stroke packaged with Maya, keyakiPark.mel, that was then converted to polygon geometry from NURBS. Although the trees were much more geometrically complex, since there was no necessity for end user to deal with loading or graphical processing of the tree in three dimensions, this was not something that needed to be worried about. More importantly, the trees appeared to not be out of place in the environment in which they were presented, a fear that arose early on when considering increasing the complexity of certain level objects. After it was established that this fear was in vain, the same process was applied to the creation of the shrubs that surround the stage and at the turning point of the road.
The grass was originally composed of a single color like the rest of the objects in the level, but the single color over such a large area proved to be inadequate to demonstrate that the object was grass. This was also the case for the use of a texture that had a random assortment of different shades of green. The final texture given to the ground plane was a texture composed of the grass brush in Photoshop

that, like the more complex variation of the tree, although more complex than the surrounding objects, was visually pleasing and was still simple enough as to not diverge from the intended overall look of the level. The only fall back to the new grass texture is that it made the walking path texture, which was once appropriate for its surroundings, too simplified compared to the surrounding grass, so a simple texture was applied to the path and this solved the issue.

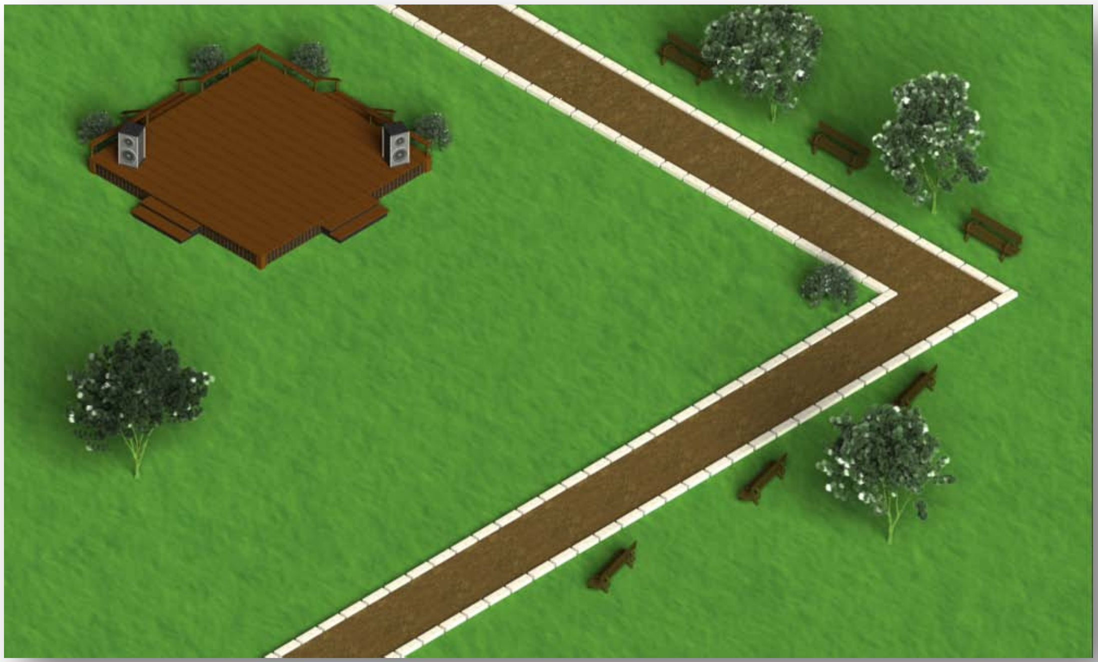


Figure 4.2:3 Final render of Park level

The sole exception to the maintaining the construction out of simple geometry is the chairs featured in the Stadium level. These chairs were made with a much higher polygon count than most objects because in a simplified for the seats could be mistaken to look something more along that lines of a beach chair. As with the trees in the park venues, increased total polygon count this produced was not an issue because of the level was going to be output in two dimensions. The only issue this raised was that it became very difficult to operate Maya effectively because of the processing power that was required to display all the chairs in the scene.

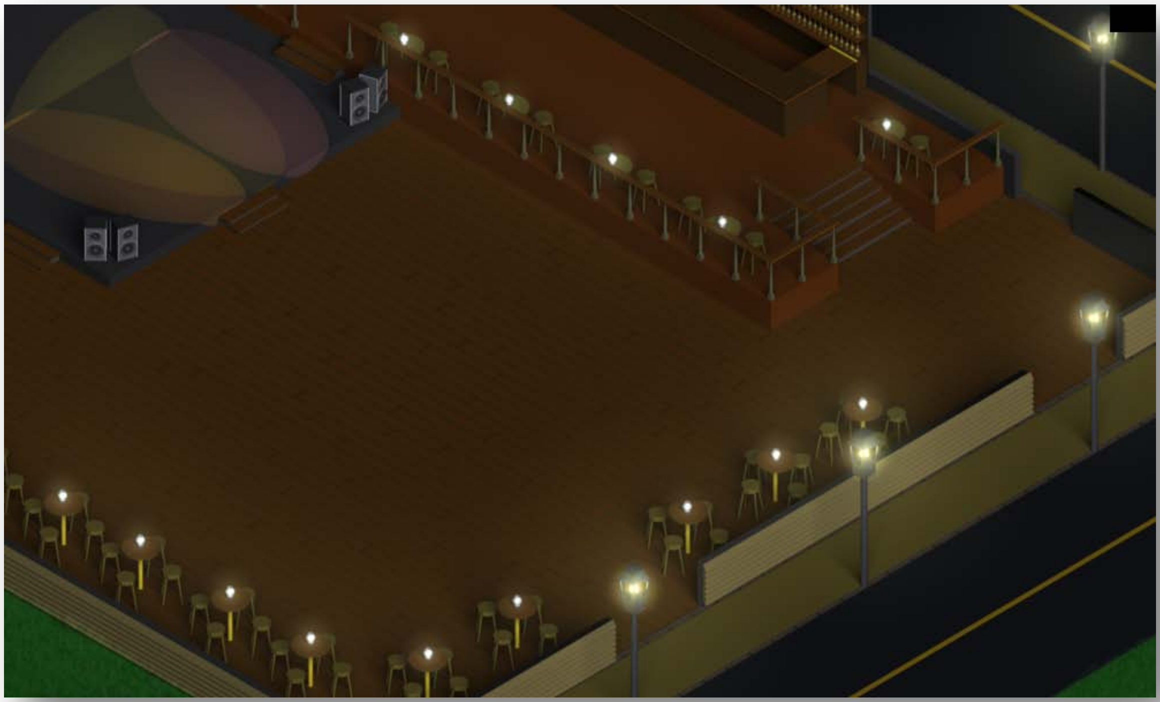


Figure 4.2:4 Final render of Bar level

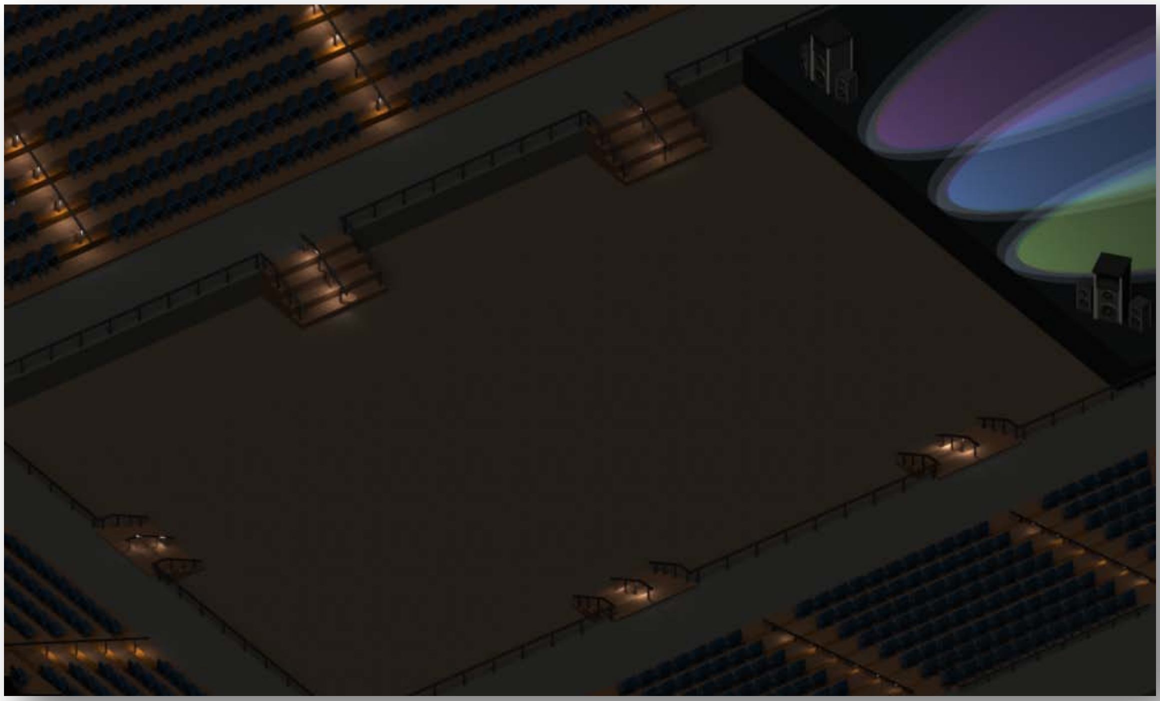


Figure 4.2:5 Final render of Arena level

4.3.Characters

From our first game design document onwards the group decided on having different types of fans that resemble genres of music. We wanted to go with fans of music genres that looked different enough to have their differences seen from far away, as they would be small sprites. Our final designs for each character gave each a predominant color, like red on the gangsta, or green on the punk, that would enable players to tell them apart at a glance.

We originally thought up a number of characters, but our final choices were the goth, the punk, and the gangsta types, as their designs made them very different than each other, so they could be picked out in a crowd. A fan who wore simple jeans and a t shirt was also considered an added as a "default" character who liked all kinds of music, just in case a player was not fond of an of our styles. A fan in business attire was also added; a tiring day at the office does not mean one would not wander over to a concert if it sounded great.

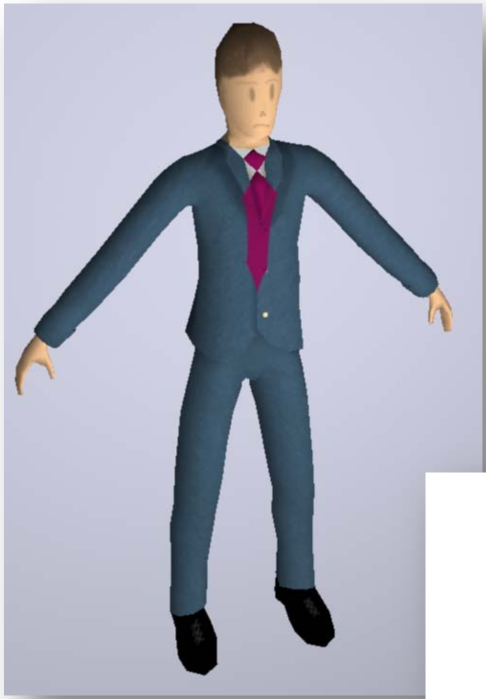
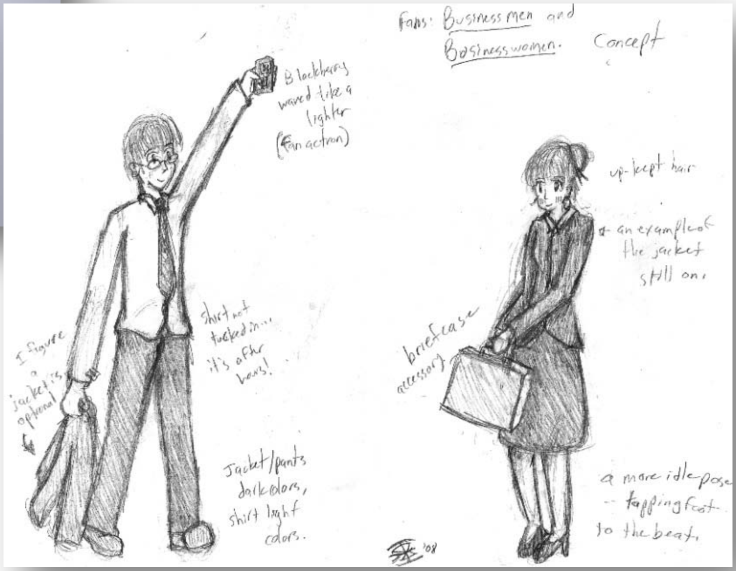


Figure 4.3:2 Render of Business character


Fans: Business men and Business women.
Concept
Blackberry waved like a lighter (fan action)
up-kept hair
an example of the jacket still on.
I figure a jacket's optional
shirt not tucked in... it's after hours!
briefcase accessory
Jacket/pants dark colors, shirt light colors.
a more idle pose -- tapping foot to the beat.
'08
Figure 4.3:1 Concept of Business character

The character models for the fans were modeled in Maya, using a model from Unreal Tournament 2003 as a base. Some small objects were also modeled that were not used in the game's final iteration. For example, a microphone stand was modeled, but Shelli threw out the idea of using it since that might create the illusion that the band was supposed to have a singer. Most of the art focus here was on modeling fan characters.

After characters were modeled, textures were created using a combination of Adobe Photoshop and the UV painting tool within Maya. Shelli used the latter to confirm the locations of sections on the UV map in Photoshop; for instance, painting green onto the front of a character's pants and then red onto the back side. The majority of the texture work was still done in Photoshop. The final sprite size was expected to be small, so Shelli chose to stick with simple textures consisting of solid colors with only minor shading added. Some elements had extra detail added in. The rocker girl had some gray speckled over her bellbottoms texture, to give them some added sparkle. Some added details would probably go unnoticed within the game, like shoelaces drawn on the shoe textures.

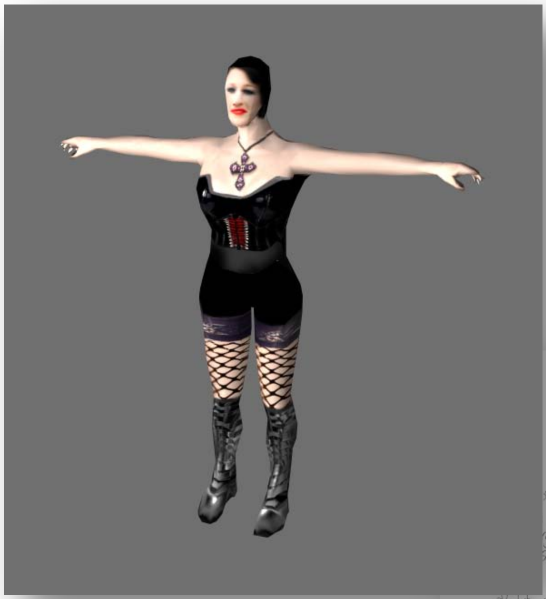
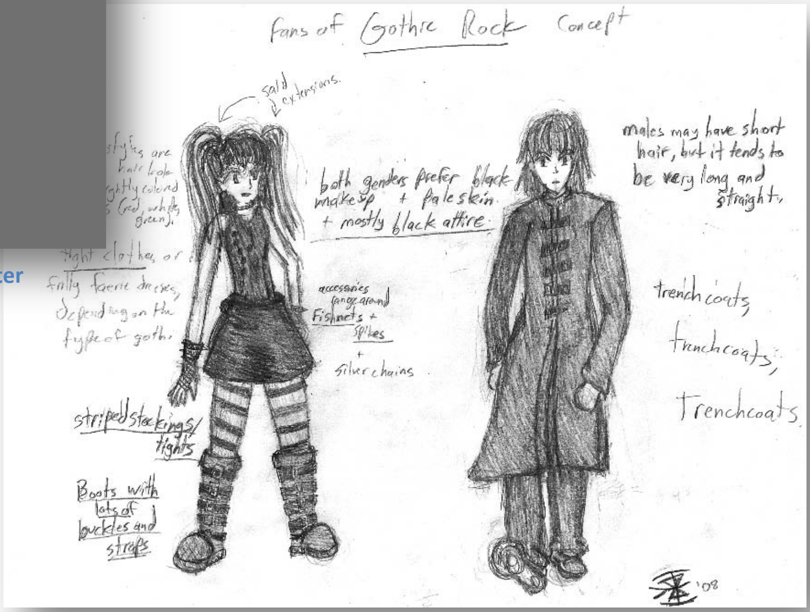


Figure 4.3:4 Render of Goth character


Fans of Gothic Rock Concept
said extensions.
styles are hair (ole) brightly colored (red, white, green).
tight clothes or frilly faerie dresses, depending on the type of goth.
both genders prefer black makeup + pale skin + mostly black attire.
accessories range around Fishnets + Spikes + silver chains
striped stockings/tights
Boots with lots of buckles and straps
males may have short hair, but it tends to be very long and straight,
trench coats,
trenchcoats,
Trenchcoats
'08
Figure 4.3:3 Concept of Goth Character

Because of the small sprite size, some areas of the model would have to be accentuated so that they were more noticeable. This problem occurred on the businessman model. Shelli's original model had his tie tucked into his suit, but it was pointed out that when zoomed out to a small size, one could not easily tell that there was a tie there at all. She remedied this by painting the rest of the tie over the suit to give it a full untucked size. It was also given a bright magenta color to stand out more.

Animations for the band were also made in Maya. Alex made animations for fans for walking and for different moods after running up to the stage area. Shelli created animations for the band members: playing guitar, playing bass, and playing drums. The various fan animations were rendered in isometric views of four different angles of front left, front right, back left, and back right; within the created levels, fans would be walking around at all of these angles. The band animations, however, were only rendered in front left and front right views, because no situation would arise where the band would be facing away from the players.

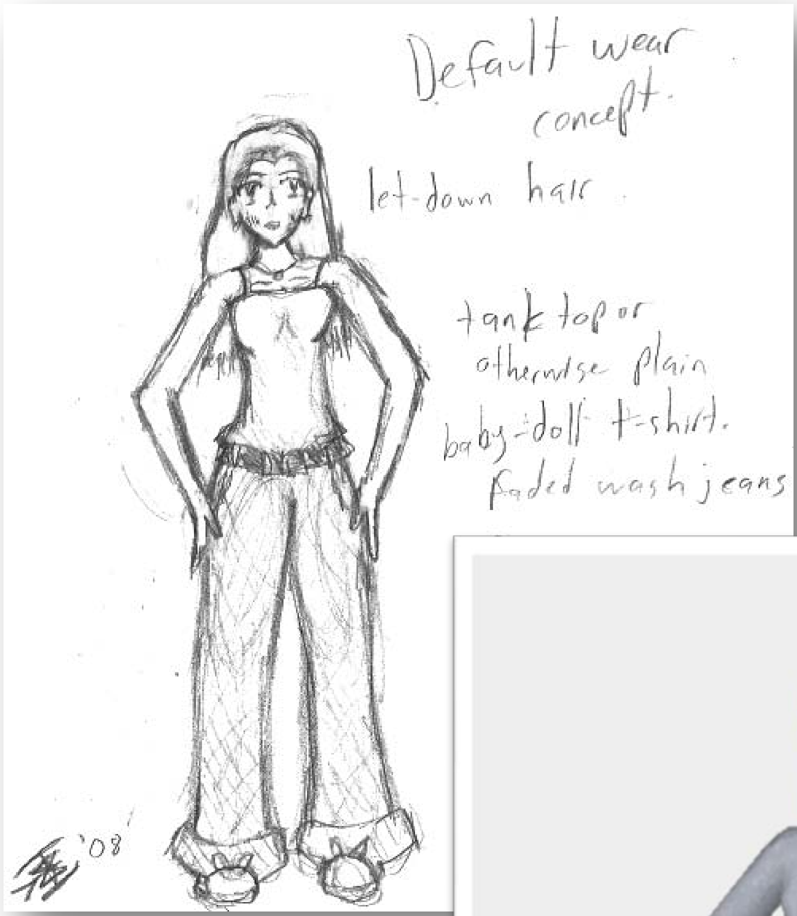


Figure 4.3:6 Concept of Default character

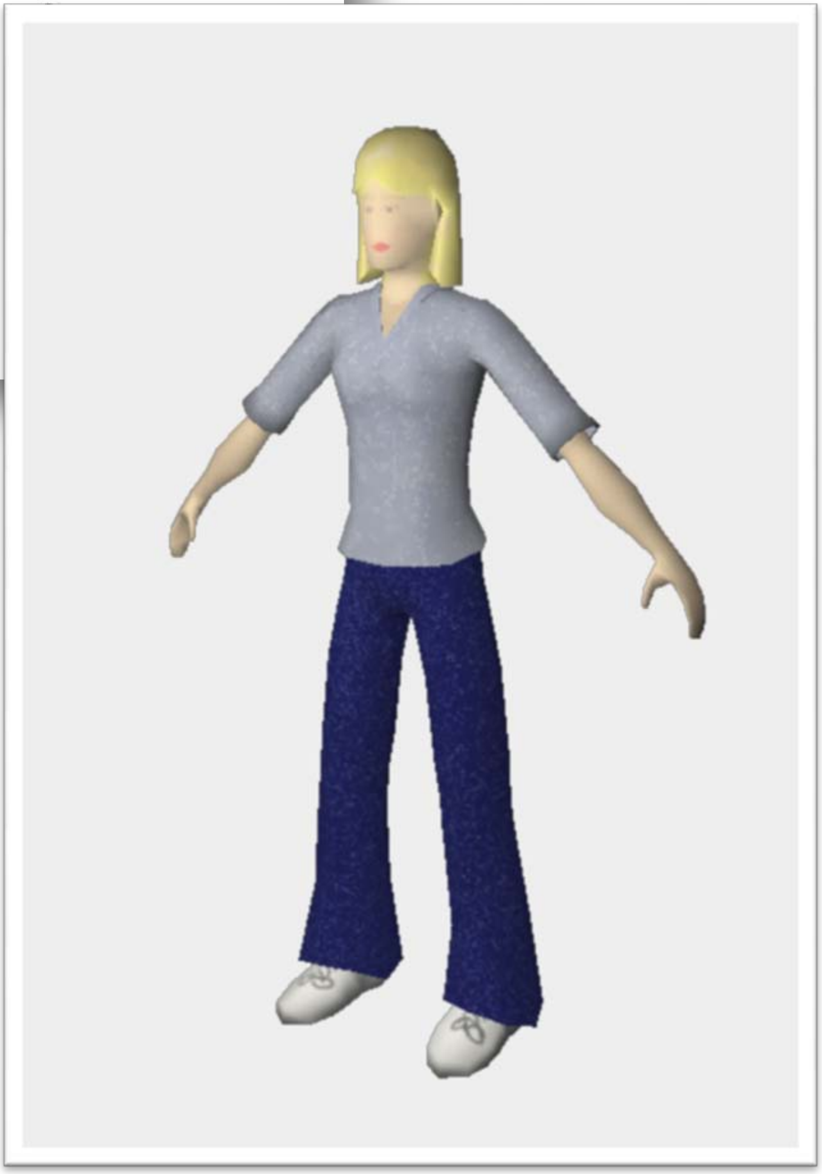


Figure 4.3:5 Render of Default character

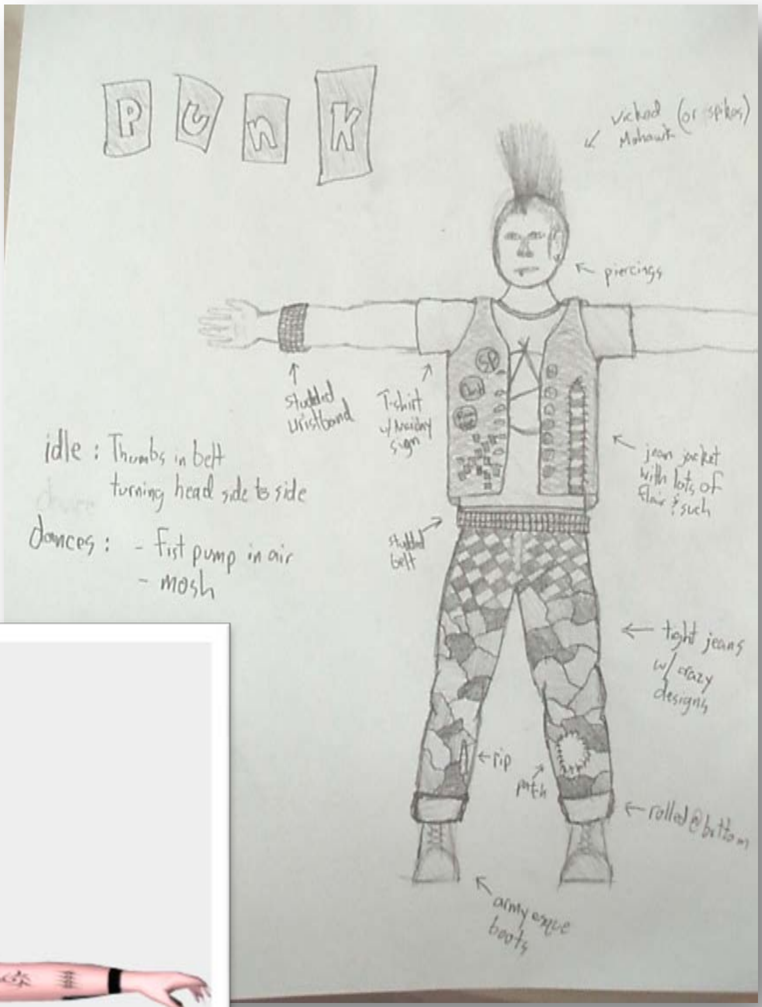

Punk
wicked Mohawk (or spikes)
piercings
studded wristband
T-shirt w/ anarchy sign
jean jacket with lots of flair & such
studded belt
tight jeans w/ crazy designs
rip
patch
rolled @ bottom
army esque boots
idle : Thumbs in belt
turning head side to side
dances : - Fist pump in air
- mosh

Figure 4.3:7 Concept of Punk Character

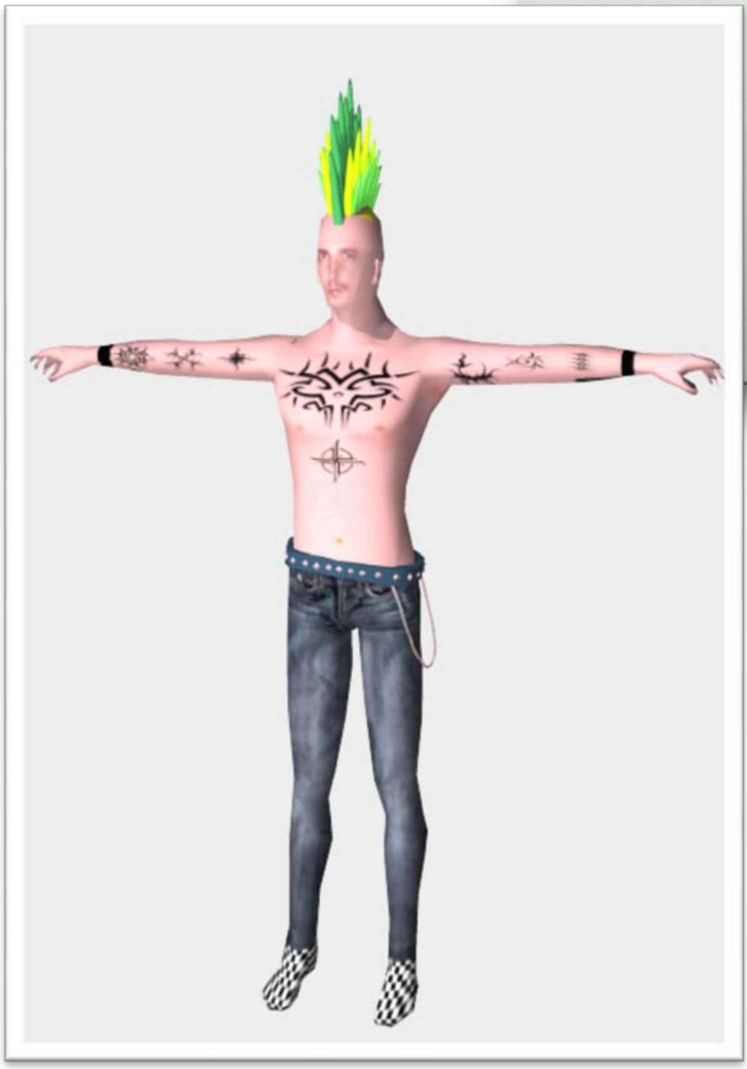


Figure 4.3:8 Render of Punk character

4.4.Code

iGotBand was coded entirely in Actionscript 2 and published for Flash Player 10. All of the code was written from scratch and many adjustments were made throughout the development process. The decision to use Actionscript 2 instead of Actionscript 3 was made on the basis that we were more familiar with Actionscript 2, and early tests showed that there were no significant improvements in performance using Actionscript 3. Specifically, keyboard handlers and the sound engine were frame-rate independent, so our choice was justified for the purposes of this game.

4.4.1. Actionscript | Main SWF file

The main SWF file contains minimal code: it imports some Flash utilities and the rest of the game code, spread out over 15 Actionscript (.as) files. This first bit imports the Tween class and Delegate utility, which are tools that we've used to create some of the functions in this game.

```
stop();
import mx.transitions.Tween;
import mx.transitions.easing.*;
import mx.utils.Delegate;
```

Then, we include 14 of the 15 Actionscript files:

```
#include "speed_vars.as"
#include "create_arrays.as"
#include "spot_array.as"
#include "fan_creator.as"
#include "fan_walker.as"
#include "fan_catcher.as"
#include "sound_importer.as"
#include "guitar_handler.as"
#include "drum_handler.as"
#include "array_functions.as"
#include "game_init.as"
#include "scene_setter.as"
#include "interface_setter.as"
```

In the following section we will discuss the purposes of each these files, as well as the strategies we used in developing them. The individual sections are organized in the same order as they are loaded into the game.

4.4.2. Actionscript | speed_vars.as

This file is loaded first to initialize the speed variables that we will use in our game loop. Basically, we are telling the game what day is it (specifically the number of milliseconds it has been since midnight, January 1st, 1970). We are also defining the current speed, which will be a fraction (1 being 100%), and the frame rate at 19 frames per second.

```
var today = new Date();
var checkTimeIn:Number = 19;
var nowTime:Number = today.getTime();
var prevTime:Number = today.getTime();
var nowSpeed:Number = 1;
var frameRate:Number = 19;
```

4.4.3. *create_arrays.as*

We must also create all of the arrays that we will be working with. The code is also commented with the intended content of each array element, to make it easier to figure out when we come back to it later.

tollArray contains every piece of information about every character that has entered the scene. This information will be tracked until the user exits the scene. This array will only expand, as there is no way to remove elements from it.

```
tollArray = new Array();
/*
* [0] status (string)
* [1] instrument (string)
* [2] note colors (array)
* [3] inFanZone
* [4] fan spot
* [5] gotoX
* [6] gotoY
*/
```

preyArray is the array that keeps track of which characters are within the "catchable" area. These are potential fans, and can be activated by playing notes. The array size dynamically adjusts via the game loop to fit the exact number of characters that it needs to.

```
preyArray = new Array();
/*
* [0] fan number (number)
* [1] instrument (string)
* [2] note colors (array)
* [3] notes played (number)
*/
```

inPlayArray contains information only about the characters that are currently visible. This information is used to "walk" characters through the scene. Like *preyArray*, this array also adjusts to fit the exact number of definite elements it has.

```
inPlayArray = new Array();
/*
* [0] fan number (number)
* [1] status (string)
```

```
*/
```

Lastly, *fanArray* is the array that will record which characters are "fans" of the player. These characters will be in the listening area of the stage, and will each have a set position. *fanArray* also adjusts its size via the game loop.

```
fanArray = new Array();
/*
* [0] fan number (number)
* [1] status (string)
* [2] spot (number)
*/
```

During early testing of the game, we only used one array (*tollArray*), which caused issues in terms of game speed. As more characters entered the scene, the array got larger and began to slow down the overall framerate of the game. The solution was to create separate arrays that are each dedicated to a specific task, as we did in the final build.

4.4.4. *spot_array.as*

We must also create an array that defines the possible spots for a fan to stand, when he or she watches the band. *spotArray* is a fixed-length array that is dynamically generated at the start of the game. It holds information about where each possible spot is located, whether or not it is occupied, and if it is, information about the fan that is standing in that spot.

```
spotArray = new Array(0);
/*
* [0] x position (number)
* [1] y position (number)
* [2] is taken? (boolean)
* [3] fan number (number)
* [4] is fan set? (boolean)
*/
```

The following function generates spots based on variables *spotRowStart, spotY, spotRowsLeft,* and *spotRowsRight,* which will be specified in the initial game code. It then randomizes the location of the spots for a more natural distribution and puts all of this information into *spotArray.*

```
function   addSpots(thisRowStart,spotY,spotRowsLeft,spotRowsRight)
{
  spotRowsTotal = spotRowsLeft+spotRowsRight-1;
  spotRow = 0;
  thisRow = 0;
  thisRowTotal = 1;
  while (spotRow < spotRowsTotal) {
    while (thisRow < thisRowTotal) {
      spotArray[spotNum]           =           new           Array
(thisRowStart+thisRow*spotSpacing*14+Math.floor(-
spotSpacing*spotRand*1+Math.random()*spotSpacing*spotRand*2+1),sp
otY+spotRow*spotSpacing*4+Math.floor(-
spotSpacing*spotRand*.6+Math.random()*spotSpacing*spotRand*1.2+1)
,0);
      thisRow++;
      spotNum++;
    }
  thisRow = 0;
  thisRowTotal++;
  if (spotRowsRight<(spotRow+2)) thisRowTotal --;
  thisRowStart -= spotSpacing*7;
  if (spotRowsLeft<(spotRow+2)) {
    thisRowStart += spotSpacing*14;
    thisRowTotal--;
  }
  spotRow ++;
}
```

4.4.5. fan_creator.as

This function will generate a new fan when called. The new fan has a random instrument icon, an incremental fan number, a random appearance (1 of 5 character archetypes) and the next set of notes loaded from the current song file.

```
createNewFan = function(this_instr, fan_num) {
      tempFanColor                                                 =
Math.floor(Math.random()*fanTypeArray.length);
      fanType = fanTypeArray[tempFanColor];
      createEmptyMovieClip("fan"+fan_num+"_mc", -fan_num-1000);
      _root["fan"+fan_num+"_mc"].createEmptyMovieClip("character_
mc", -1);
      loadMovie("characters/fans/"+fanType+".swf",
"fan"+fan_num+"_mc.character_mc");
      _root["fan"+fan_num+"_mc"].createEmptyMovieClip("healthbar_
mc", 1);
      _root["fan"+fan_num+"_mc"].healthbar_mc._y = -60;
      _root["fan"+fan_num+"_mc"].healthbar_mc._x = 6;
      tempNoteNum = 0;
      if (guitarArray[guitarNextNoteNum][0].indexOf(".") != -1) {
            guitarChordArray[guitarChordArray.length]     =     new
Array(guitarArray[guitarNextNoteNum][0].split("."), fan_num);
            guitarNextNoteNum++;
      }
      guitarBreakDown                                              =
guitarArray[guitarNextNoteNum][0].split("");
      tempNoteCount = guitarBreakDown.length;
      tollArray[fan_num]   =   new   Array("walking",   this_instr,
guitarBreakDown, inFanZone);
      while (tempNoteNum < tempNoteCount) {
            tempNoteColor                                          =
colorArray[guitarBreakDown[tempNoteNum]];

      _root["fan"+fan_num+"_mc"].healthbar_mc.attachMovie("health
_"+tempNoteColor,                       "health"+tempNoteNum+"_mc",
_root["fan"+fan_num+"_mc"].healthbar_mc.getNextHighestDepth(),
{_x:tempNoteNum*18, _y:0});
            tempNoteNum++;
      }
      _root["fan"+fan_num+"_mc"].healthbar_mc.attachMovie("health
_"+this_instr,                                          "instr_mc",
_root["fan"+fan_num+"_mc"].healthbar_mc.getNextHighestDepth(),
{_x:-20, _y:0});
      _root["fan"+fan_num+"_mc"].healthbar_mc._x       =       14-
tempNoteCount*9;
      _root["fan"+fan_num+"_mc"]._x                              =
fanStartX+Math.floor(Math.random()*fanPathWidth);
      _root["fan"+fan_num+"_mc"]._y = fanStartY;
      inPlayArray[inPlayArray.length]    =    new    Array(fan_num,
"walking");
}
```

On character creation, it will also set the new character to "walking" mode, which will be recognized by *fan_walker.as*.

4.4.6. fan_walker.as

This bit is what makes a character walk through the scene. It checks where the character is, and moves him or her in the right direction. Also, when a character leaves the play area, it deletes the appropriate array element and closes the gap by splicing the array at the location of that element.

```
walkFan = function(fan) {
  if (_root["fan"+fan+"_mc"]._y < 262) {
    _root["fan"+fan+"_mc"]._x += (7/2)/nowSpeed;
  }
  else {
    _root["fan"+fan+"_mc"].swapDepths(90000-fan);
    _root["fan"+fan+"_mc"].character_mc.walking_mc
    .gotoAndStop("fl");
    _root["fan"+fan+"_mc"]._x -= (6.8/2)/nowSpeed;
    if (_root["fan"+fan+"_mc"]._y > 340
    && inPlayArray[i][1] != "prey") {
      inPlayArray[i][1] = "prey";
      addPrey(fan);
    }
    if (_root["fan"+fan+"_mc"]._y > 630
    && inPlayArray[i][1] != "walking") {
      inPlayArray[i][1] = "walking";
      _root["fan"+fan+"_mc"].healthbar_mc.instr_mc
      .gotoAndStop("unlit");
      removePrey(fan);
    }
  }
  _root["fan"+fan+"_mc"]._y += (4/2)/nowSpeed;
  if (_root["fan"+fan+"_mc"]._y >= 700) {
  _root["fan"+fan+"_mc"].unloadMovie();
  for (i=0;i<inPlayArray.length;i++) {
    if (inPlayArray[i][0] == fan) delete inPlayArray[i];
      break;
    }
    stripUndefined(inPlayArray);
  }
}
```

The plural function *walkFans* goes through the length of *inPlayArray* and sends the numbers to *walkFan*.

```
walkFans = function() {
      for (i=0;i<inPlayArray.length;i++) {
            walkFan(inPlayArray[i][0]);
      }
}
```

approachFan is called when a fan is "caught," and starts approaching the stage. This function moves the fan into the grass area to indicate that he or she has been caught.

```
approachFan = function(fan) {
            if              (_root["fan"+fan+"_mc"]._x            +
_root["fan"+fan+"_mc"]._y*1.7 > 1156) {
                  _root["fan"+fan+"_mc"]._x -= 14;
                  _root["fan"+fan+"_mc"]._y -= 8;
            }
            else {
                  for (i=0;i<fanArray.length;i++) {
                        if (fanArray[i][0] == fan) {
                              fanArray[i][1] = "setting";
                              setFan(fan, i);
                        }
                  }
            }
}
```

Similarly, *approchFans* moves every fan who is currently approaching by calling the *approachFan* function for each of those fans.

```
approachFans = function() {
      for (a=0;a<fanArray.length;a++) {
            if        (fanArray[a][1]        ==        "approach")
approachFan(fanArray[a][0]);
      }
}
```

setFan is called directly after the fan reaches the grass position indicated by *approachFan*. It creates a path for the fan to follow in order to reach his or her final point on the screen.

```
setFan = function(fan, arrayloc) {
      fanCount++;
      _root["fan"+fan+"_mc"].setX        =        new        Tween
(_root["fan"+fan+"_mc"],           "_x",           None.easeNone,
_root["fan"+fan+"_mc"]._x,    spotArray[fanArray[arrayloc][2]][0],
10, false);
      _root["fan"+fan+"_mc"].setY        =        new        Tween
(_root["fan"+fan+"_mc"],           "_y",           None.easeNone,
_root["fan"+fan+"_mc"]._y,    spotArray[fanArray[arrayloc][2]][1],
10, false);
      _root["fan"+fan+"_mc"].setX.onMotionFinished               =
Delegate.create(_root["fan"+fan+"_mc"], landFan);
}
```

The *landFan* function is called when a fan reaches his or her final destination point. This function also tells the fan to go to the "dancejump" frame, which contains an animation of him or her jumping to the music.

```
landFan = function() {
      this.character_mc.gotoAndStop("dancejump");
      this.swapDepths(100000+10*Math.floor(this._y)-fanCount);
      if               (this._x                <               234)
this.character_mc.dancejump_mc.gotoAndStop("br");
}
```

4.4.7. fan_catcher.as

The *getBored* function is called every random interval, and it assigns one of the current fans to the "bored" status. That fan will then go to the frame which contains an "exclaim" animation.

```
function getBored (spot) {
      _root["fan"+spotArray[spot][3]+"_mc"].fanStatus = "bored";
      exclaimStyle = Math.floor(Math.random()*5);
      if (exclaimStyle == 0) {

      _root["fan"+spotArray[spot][3]+"_mc"].gotoAndStop("exclaimc
rossarms");
      }
      else if (exclaimStyle == 1) {

      _root["fan"+spotArray[spot][3]+"_mc"].gotoAndStop("exclaimf
lip");
      }
      else if (exclaimStyle == 2) {

      _root["fan"+spotArray[spot][3]+"_mc"].gotoAndStop("exclaimj
ump");
      }
      else if (exclaimStyle == 3) {

      _root["fan"+spotArray[spot][3]+"_mc"].gotoAndStop("exclaimo
hsheeit");
      }
      else {
```

```
      _root["fan"+spotArray[spot][3]+"_mc"].gotoAndStop("exclaims
adness");
      }
      _root["fan"+spotArray[spot][3]+"_mc"].exclaim_mc.gotoAndSto
p(_root["fan"+spotArray[spot][3]+"_mc"].fanDir);
}
```

When the player plays a note on the guitar, the *catchNote* function is called, which searches for a fan that has an applicable note. The function prioritizes fans who have one or more notes that have already been played, then fans who are the closest to the exit (or spawned first).

```
catchNote = function(this_instr, dominant, note) {
      for (i=0;i<preyArray.length;i++) {
            if (preyArray[i][1] == this_instr) {
                  if (preyArray[i][3] > 0) {
                        if
(parseInt(preyArray[i][2][preyArray[i][3]]) == note) {
                        killNote(i);
                        return;
                        }
                  }
            }
      }
      //resetNotes("guitar");
      for (i=0;i<preyArray.length;i++) {
            if (preyArray[i][1] == this_instr) {
                  if   (parseInt(preyArray[i][2][preyArray[i][3]])
== note && dominant) {
                        killNote(i);
                        return;
                  }
            }
      }
}
```

If the note that was played matches the next note of one of the fans, *killNote* will remove it.

```
killNote = function(prey) {
      _root["fan"+preyArray[prey][0]+"_mc"].healthbar_mc.instr_mc
.gotoAndStop("lit");
      _root["fan"+preyArray[prey][0]+"_mc"].healthbar_mc["health"
+preyArray[prey][3]+"_mc"]._alpha = 0;//gotoAndStop("lit");
      preyArray[prey][3]++;
      notes_left        =        preyArray[prey][2].length        -
preyArray[prey][3];
      if (notes_left <= 0) {
            //trace("caught: "+preyArray[prey][0]);

      _root["fan"+preyArray[prey][0]+"_mc"].healthbar_mc.instr_mc
._alpha = 0;
            catchFan(preyArray[prey][0]);
      }
      else {
            j = 0;
            for
(i=preyArray[prey][3];i<preyArray[prey][2].length;i++){

      _root["fan"+preyArray[prey][0]+"_mc"].healthbar_mc["health"
+i+"_mc"]._x = j*18;
            j++;
            }
            _root["fan"+preyArray[prey][0]+"_mc"].healthbar_mc._x
= 14-notes_left*9;
      }
}
```

When all notes of a fan have been played, the *catchFan* function is called.

```
catchFan = function (fan) {
      for (g=0;g<preyArray.length;g++) {
            if (preyArray[g][0] == fan){
                  findSpot(fan);
            }
      }
}
```

When a fan is caught, *findSpot* searches for a spot within *spotArray* to assign to that fan.

```
findSpot = function(fan) {
      spotSearching = 1;
      newSpot = null;
      if (spotArray.length > fanCount) {
            while (spotSearching) {
                  newSpot                                        =
Math.floor(Math.random()*spotArray.length);
                  spotSearching = spotArray[newSpot][2];
            }
            spotArray[newSpot][2] = 1;
            fanArray[fanArray.length]     =     new     Array(fan,
"approach", newSpot);

      _root["fan"+fan+"_mc"].character_mc.walking_mc.gotoAndStop(
"bl");
            removePrey(fan);
            removeWalker(fan);
      }
}
```

4.4.8. *sound_importer.as*

When the game is first loaded, *sound_importer.as* defines the notes that will be imported into the game.

```
// DEFINE NOTES
var guitar_note0:String;
var guitar_note1:String;
var guitar_note2:String;
var guitar_note3:String;
var guitar_note4:String;
var guitar_note5:String;
var drum_note1:String = "samples/drum/cymbal.mp3";
var drum_note2:String = "samples/drum/snare.mp3";
var drum_note3:String = "samples/drum/hitom.mp3";
var drum_note4:String = "samples/drum/lotom.mp3";
var drum_note5:String = "samples/drum/kick.mp3";
```

It then creates channels for each of the 6 guitar notes as well as the drum pads. The drum pads do not affect game play in this release, but they can be used to play drum sounds.

```
//  CREATE CHANNELS
var guitar_ch0:Sound = new Sound();
var guitar_ch1:Sound = new Sound();
var guitar_ch2:Sound = new Sound();
var guitar_ch3:Sound = new Sound();
var guitar_ch4:Sound = new Sound();
var guitar_ch5:Sound = new Sound();
var drum_ch1:Sound = new Sound();
var drum_ch2:Sound = new Sound();
var drum_ch3:Sound = new Sound();
var drum_ch4:Sound = new Sound();
var drum_ch5:Sound = new Sound();
var bgm:Sound = new Sound();
```

And finally, the sounds get imported into the game.

```
// IMPORT SAMPLES
importSounds  =  function(for_instr,  instr_tone,  sound0,  sound1,
sound2, sound3, sound4, sound5) {
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound0+".mp3";
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound1+".mp3";
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound2+".mp3";
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound3+".mp3";
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound4+".mp3";
//    _root[for_instr+"note0"]                                  =
"samples/"+for_instr+"/"+instr_tone+"/"+sound5+".mp3";
      _root[for_instr+"_ch0"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound0+".mp3");
      _root[for_instr+"_ch1"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound1+".mp3");
      _root[for_instr+"_ch2"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound2+".mp3");
```

```
      _root[for_instr+"_ch3"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound3+".mp3");
      _root[for_instr+"_ch4"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound4+".mp3");
      _root[for_instr+"_ch5"].loadSound("samples/"+for_instr+"/"+
instr_tone+"/"+sound5+".mp3");
}
drum_ch1.loadSound(drum_note1);
drum_ch2.loadSound(drum_note2);
drum_ch3.loadSound(drum_note3);
drum_ch4.loadSound(drum_note4);
drum_ch5.loadSound(drum_note5);
```

4.4.9. *guitar_handler.as*

The Actionscript file *guitar_handler.as* contains information about how to handle the guitar controller actions. The first two objects it creates are *guitar_pressed* and *guitar_playing. guitar_pressed* is a boolean that is true when a key is being held down, and *guitar_playing* is true when a note is currently playing.

```
var guitar_pressed:Object = {};
var guitar_playing:Object = {};

// DEFAULT OPEN NOTE IS ALWAYS PRESSED. ALWAYS!!!
guitar_pressed[0] = 1;
```

This bit of code creates an array to define each guitar button.

```
guitar_key = new Array();
guitar_key[0] = guitar_key_white;
guitar_key[1] = guitar_key_green;
guitar_key[2] = guitar_key_red;
guitar_key[3] = guitar_key_yellow;
guitar_key[4] = guitar_key_blue;
guitar_key[5] = guitar_key_orange;
guitar_key[6] = guitar_key_up;
guitar_key[7] = guitar_key_down;
guitar_key[8] = guitar_key_prev;
guitar_key[9] = guitar_key_next;
```

Then it sets the key code for each button on the guitar.

```
guitar_key_up = 38;
guitar_key_down = 40;
guitar_key_green = 49;
guitar_key_red = 50;
guitar_key_yellow = 51;
guitar_key_blue = 52;
guitar_key_orange = 53;
guitar_key_prev = Key.LEFT;
guitar_key_next = Key.RIGHT;
```

Since the mp3 files for the guitar notes contain a 50 millisecond period of silence (inevitable for mp3 files), we define the offset start position to be .05 seconds into the sample. Also, we set the guitar hammer volume to be 90% of the regular note. This means that hammering a note will be slightly quieter than strumming it.

```
guitar_pick_offset = .05;
guitar_hammer_offset = .05;
guitar_hammer_volume = 90;
```

playGuitarNote is called when the user plays a note,

```
playGuitarNote = function(picked, note) {
      catchNote("guitar", picked, note);
      if (picked) {
            _root["guitar_ch"+note].setVolume(100);
            _root["guitar_ch"+note].start(guitar_pick_offset);
      }
      else {

      _root["guitar_ch"+note].setVolume(guitar_hammer_volume);
            _root["guitar_ch"+note].start(guitar_hammer_offset);
      }
      guitar_playing[note] = 1;
}
```

and stop*GuitarNote* is called when the user plays a different note, or lets go of all notes.

```
stopGuitarNote = function(note) {
      _root["guitar_ch"+note].stop();
}
```

The *playGuitarHighest* function finds the highest note that is currently being pressed and plays it.

```
playGuitarHighest = function(picked) {
      stopGuitarNotes();
      for(n=5;n>=1;n--) {
            if (guitar_pressed[n]){
                  playGuitarNote(picked, n)
                  return;
            }
      }
      if (picked) {
            playGuitarNote(picked, 0)
            return;
      }
}
```

We define functions for when a note is picked, hammered and pringed.

```
pickGuitarNote = function() {
      guitarIsHot = 2;
      playGuitarHighest(1);
}
hammerGuitarNote = function(note) {
      guitar_pressed[note] = 1;
      if (guitar_playing[note]) return;
      for(n=5;n>note;n--) {
            if (guitar_pressed[n]) return;
      }
      guitar_ch0.stop();
      for(n=0;n<note;n++) {
            if (guitar_playing[n] == 1) {
                  playGuitarHighest(0);
                  return;
            }
            if(guitarIsHot > 0) playGuitarHighest(0);
      }
      // guitar_pressed_num = note;
}
pringGuitarNote = function(note) {
      stopGuitarNote(note);
      guitar_pressed[note] = 0;
      for(n=5;n>note;n--) {
            if (guitar_pressed[n]) return;
      }
      if (guitarIsHot > 0) playGuitarHighest(0);
      for(n=5;n>0;n--) {
            if (guitar_playing[n] == 1) {
                  guitarIsHot = 2;
                  playGuitarHighest(0);
                  return;
            }
      }
}
```

```
stopGuitarNotes = function() {
      for(n=0;n<=5;n++) {
            stopGuitarNote(n);
            guitar_playing[n] = 0;
      }
}
```

guitarKeyListener constantly checks for notes being held.

```
guitarKeyListener = new Object();
guitarKeyListener.onKeyDown = function() {
      // default open note pressed
      // guitar_pressed_num = 0;
      // get key code for pressed key
      e = Key.getCode();
      // if key is a note button, set guitar_pressed for that
note to 1
      if(e == guitar_key_green || e == guitar_key_red || e ==
guitar_key_yellow || e == guitar_key_blue || e ==
guitar_key_orange){
            if(e == guitar_key_green) hammerGuitarNote(1);
            if(e == guitar_key_red) hammerGuitarNote(2);
            if(e == guitar_key_yellow) hammerGuitarNote(3);
            if(e == guitar_key_blue) hammerGuitarNote(4);
            if(e == guitar_key_orange) hammerGuitarNote(5);
            note0.stop();
      }
      // play sound when spacebar is pressed
      if(e == guitar_key_up || e == guitar_key_down) {
            //gp = 0;
            pickGuitarNote();
            /*m = 0;
            for(n=1;n<=5;n++) {
                  if(guitar_pressed[n]) {
                        playGuitarNote(n);
                        m++;
                  }
            }
            if (m == 0) playGuitarNote(0);*/
      }
      if (Key.getCode() == Key.LEFT){
            prevGuitarTone();
      }
      if (Key.getCode() == Key.RIGHT){
            nextGuitarTone();
      }
}
```

```
// STOP NOTES THAT HAVE BEEN RELEASED
guitarKeyListener.onKeyUp = function() {
      e = Key.getCode();
      if(e == guitar_key_green || e == guitar_key_red || e ==
guitar_key_yellow || e == guitar_key_blue || e ==
guitar_key_orange){
            //if(guitar_pressed[e]) return;
            if(e == guitar_key_green) pringGuitarNote(1);
            if(e == guitar_key_red) pringGuitarNote(2);
            if(e == guitar_key_yellow) pringGuitarNote(3);
            if(e == guitar_key_blue) pringGuitarNote(4);
            if(e == guitar_key_orange) pringGuitarNote(5);
      }
}
Key.addListener(guitarKeyListener);
```

4.4.10. arrayfunctions.as

We define a function called *stripUndefined*, which checks an array for undefined elements and closes these unused spaces by splicing the array at those points.

```
function stripUndefined(arr:Array) {
      for(var i=0;i<arr.length;i++){
            if(arr[i]==undefined){
                  arr.splice(i,1);
            }
      }
}
```

The *addPrey* function adds a new fan to *preyArray*.

```
function addPrey(fan_num) {
      preyArray[preyArray.length]     =     new     Array(fan_num,
tollArray[fan_num][1], tollArray[fan_num][2], 0);
      if (guitarChordArray[0][1] == fan_num) {
            importSounds("guitar",                      "distortion",
guitarChordArray[0][0][0],              guitarChordArray[0][0][1],
guitarChordArray[0][0][2],              guitarChordArray[0][0][3],
guitarChordArray[0][0][4], guitarChordArray[0][0][5]);
            delete guitarChordArray[0];
            stripUndefined(guitarChordArray);
      }
}
```

Similarly, remove*Prey* removes a fan from *preyArray*.

```
function removePrey(fan_num) {
      for (p=0;p<preyArray.length;p++) {
            if (preyArray[p][0] == fan_num) {
                  delete preyArray[p];
                  break;
            }
      }
      stripUndefined(preyArray);
}
```

removeWalker removes a fan from the game entirely.

```
function removeWalker(fan_num) {
      for (k=0;k<inPlayArray.length;k++) {
            if (inPlayArray[k][0] == fan_num) {
                  delete inPlayArray[k];
                  break;
            }
      }
      stripUndefined(inPlayArray);
}
```

4.4.11. game_init.as

game_init.as sets all of the initial values in the game.

```
var frameCount:Number = new Number();
frameCount = 0;
var i:Number = new Number();
i = 1;
var t:Number = new Number();
t = 0;
var gf:Number = new Number();
gf = 0;
var b:Number = new Number();
b = 0;
var guitarIsHot:Number = new Number();
guitarIsHot = 0;
var guitarFanNext:Number = new Number();
guitarFanNext = 1;
var creationIndex:Number = new Number();
creationIndex = 20;
var boredFansEvery:Number = new Number();
boredFansEvery = 200;
var boredIndex:Number = new Number();
boredIndex = 50;
var tollCount:Number = new Number();
tollCount = 0;
var fanSpeed:Number = new Number();
fanSpeed = 1;
var approachSpeed:Number = new Number();
approachSpeed = 2;
var showInfo:Boolean = new Boolean();
showInfo = false;
var fanColor:Number = new Number(); // 1 through 5, representing
the buttons from red to orange.
fanColor = 0;
var fanTap:Boolean = new Boolean(); // boolean, denotes whether
or not a fan is tap-able
fanTap = false;
var tapDat:String = new String();
tapDat = "";
var p:Number = new Number();
p = 0;
var pf:Number = new Number();
pf = 0;
var noteMin:Number = new Number();
noteMin = 5;
var noteMax:Number = new Number();
noteMax = 5;
var guitarNextNoteNum:Number = new Number();
guitarNextNoteNum = 0;
var songTempo:Number = new Number();
songTempo = 120;
```

```
colorArray = new Array("white", "green", "red", "yellow", "blue",
"orange");
fanTypeArray  =  new  Array("punk",  "gangsta",  "girl",  "man",
"goth");
instrArray = new Array("guitar", "bass", "drum");
tempSongArray = new Array();
songArray = new Array();
```

It also creates a new LoadVars for *songData*, the file that is read in from an external igb. file.

```
songData = new LoadVars();
songData.onLoad = function() {
      song_title_txt.text = this.songTitle;
      songTempo = parseInt(this.tempo);
      thisScene = this.scene;
      for(j = 0; j < instrArray.length; j++) {
            _root[instrArray[j]+"Notes"] = new Array();
            _root[instrArray[j]+"Notes"] = this.guitar;
            tempSongArray                                       =
_root[instrArray[j]+"Notes"].split("\n");
            importSounds("guitar",                  "distortion",
tempSongArray[0].split(".")[0],    tempSongArray[0].split(".")[1],
tempSongArray[0].split(".")[2],    tempSongArray[0].split(".")[3],
tempSongArray[0].split(".")[4],
tempSongArray[0].split(".")[5].split(":")[0]);
            delete tempSongArray[0];
            stripUndefined(tempSongArray);
            _root[instrArray[j]+"Array"] = new Array();
            for(i=0;i<tempSongArray.length;i++) {
                  _root[instrArray[j]+"Array"][i]               =
tempSongArray[i].split(":");
            }
      }
      //importSounds("guitar",  "distortion",  "c5",  "d5",  "eb5",
"f5", "g5", "ab5");
      bgm.loadSound("samples/tracks/"+currentSong+".mp3", true);
      bgm.setVolume(80);
      #include "game_loop.as"
};
songData.load("songdata/"+currentSong+".igb");
```

4.4.12. game_loop.as

The game loop is called every frame. It checks the current time in milliseconds and adjusts the game speed to avoid latency between the sound and visuals. It also increments the important game variables.

```
onEnterFrame = gameLoop;
function gameLoop():Void {
      // check time
      if(checkTimeIn <= 0) {
         today = new Date();
         prevTime = nowTime;
         nowTime = today.getTime();
```

```
          nowSpeed = int((1000/(nowTime - prevTime))*10000)/10000;
         // nowLatency = 1000/(nowSpeed*19);
          checkTimeIn = frameRate;
          speed_txt.text = nowSpeed;
      }
      checkTimeIn--;
      // create a new person
      if (gf >= guitarFanNext) {
            if (guitarArray[guitarNextNoteNum] == null) {}
            else {
                  createNewFan("guitar", tollCount);
                  tollCount++;
                  var thisNoteLength:Number = new Number;
                  thisNoteLength                                  =
parseInt(guitarArray[guitarNextNoteNum][1]);
                  guitarFanNext                                  +=
thisNoteLength*(1140/songTempo);
                  guitarNextNoteNum++;
            }
      }
      walkFans();
      approachFans();
      // get bored
      frame_count_txt.text = frameCount;
      frameCount++;
      if (guitarIsHot > 0) guitarIsHot--;
      else guitarIsHot = 0;
      t++;
      gf += 1/nowSpeed;
      b++;
      i=0;
      p=0;
      pf=0;
      gp++;
}
```

4.4.13. scene_setter.as

The first section of this code tells the band members which direction to face at the start of the game.

```
guitarist_mc.gotoAndStop("fr");
drummer_mc.gotoAndStop("fr");
bassist_mc.gotoAndStop("fl");
```

Then we move the static scenery objects to appropriate depths.

```
tree_mc.swapDepths(110000);
fence1_mc.swapDepths(83401);
fence2_mc.swapDepths(52801);
```

We also define where the fans start when they enter the scene.

```
var fanStartX:Number = new Number();
fanStartX = 332;
var fanStartY:Number = new Number();
fanStartY = -25;
var fanPathWidth:Number = new Number();
fanPathWidth = 60;
```

And finally we create spots for *spotArray*.

```
addSpots(88,204,3,5);
addSpots(424,156,12,9);
```

4.5.Website

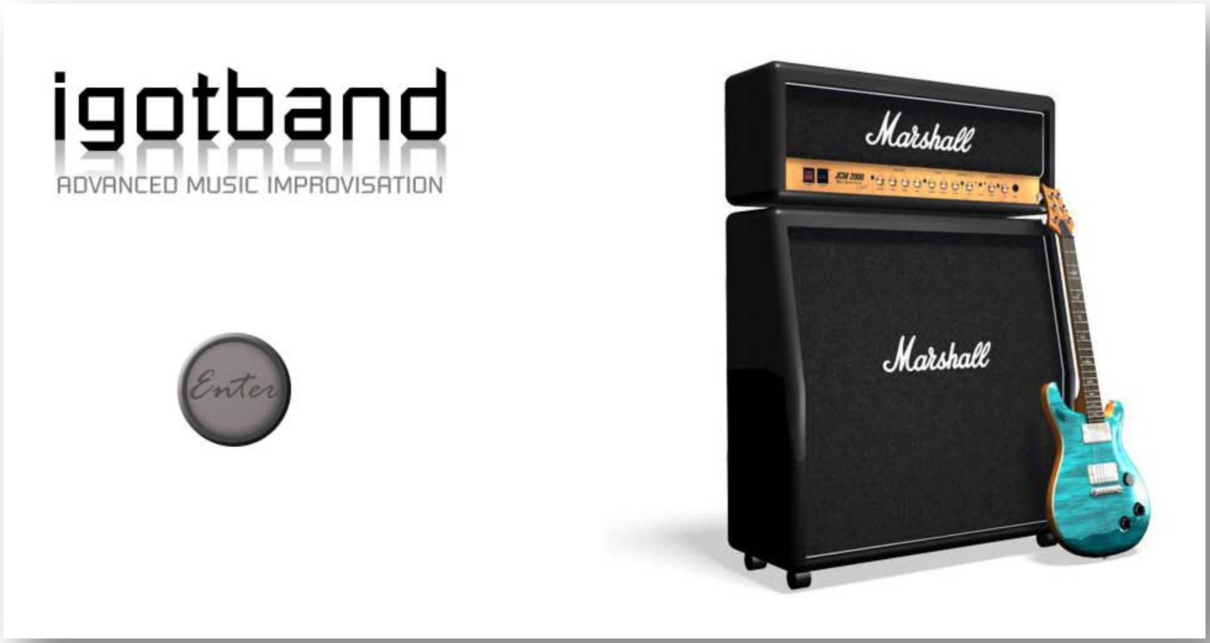
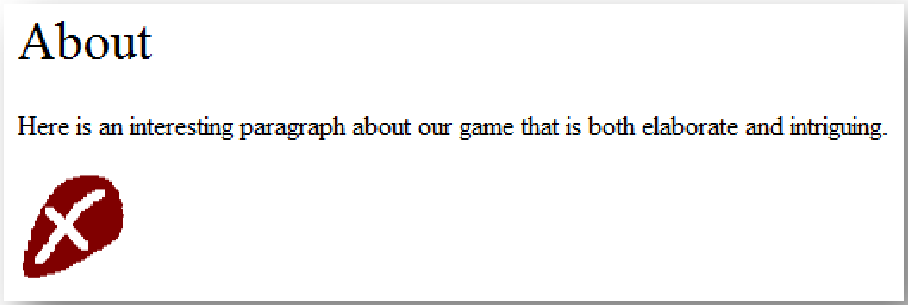


Figure 4.5:1 Index page of iGotBand.com

Both the first and second iterations of the website were designed and coded by Tim Cushman. The first iteration of the website was written in notepad, and had almost no flow from one page to another because the original design had all pages go through the home page (the page with the large render of the guitar in the upper left). Even thought the website was not part of the actual game project, a good amount of time was put into it because it served as a stage from which we would display some high quality renders that were made, as well as the point from which we would distribute the game to those interested in playing it.

```
<html>
<head>
</head>
<body>
<p>
<font size="+3">
About
</font>
<br><br>
Here is an interesting paragraph about our game that is both elaborate and intriguing.
</p>
</div>
<div id="contentRight">
<div id="buttonDiv">
<a href="iGotBand_MAIN.html"
        onMouseOver="movr(1);return true;"
        onMouseOut="mout(1);return true;"
><img name=img1 width=64 height=64 border=0 alt=""src="images/placeholder_pick.png">
</a>
</div>
</body>
</html>
```

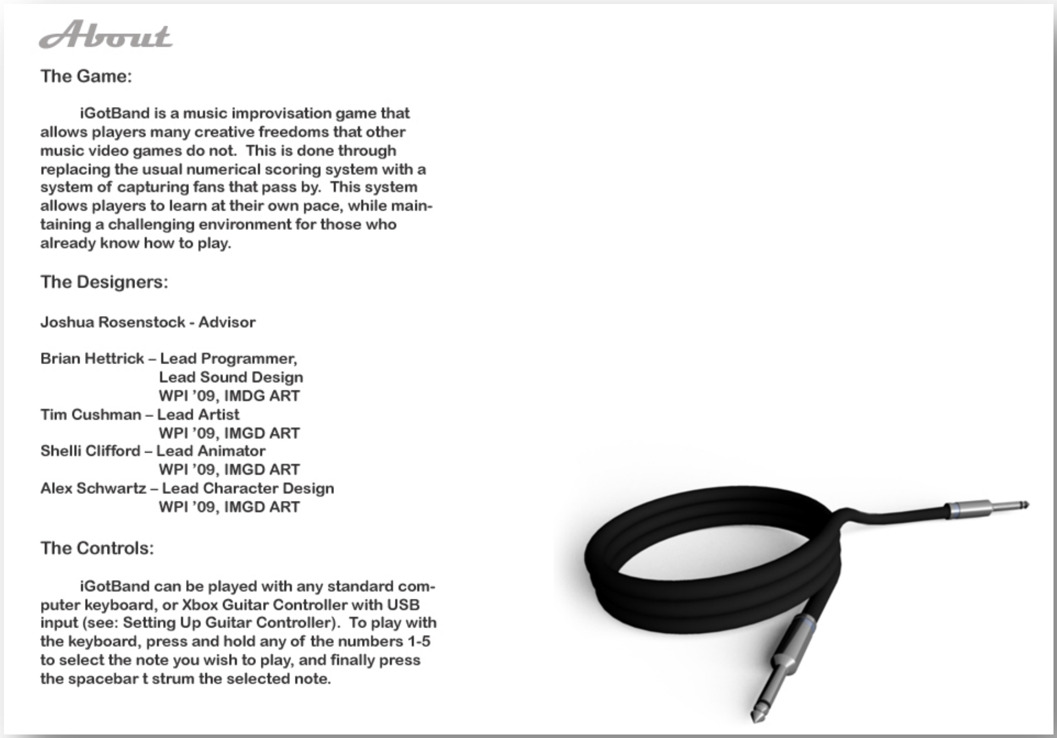


About

Here is an interesting paragraph about our game that is both elaborate and intriguing.

Figure 4.5:2 Design of website hardcoded in Notepad

After the enlightenment as to the existence of Dreamweaver, all efforts were made to convert the existing site into something that could be crafted by someone who is more artistically inclined. This was particularly important since a large part of the planning of the game was to have it be easily accessible to anyone who wants to play it, which was some of the reasoning behind have the game being coded in Flash. As seen in the code below I used frames to operate the navigation bar.



About

The Game:

iGotBand is a music improvisation game that allows players many creative freedoms that other music video games do not. This is done through replacing the usual numerical scoring system with a system of capturing fans that pass by. This system allows players to learn at their own pace, while maintaining a challenging environment for those who already know how to play.

The Designers:

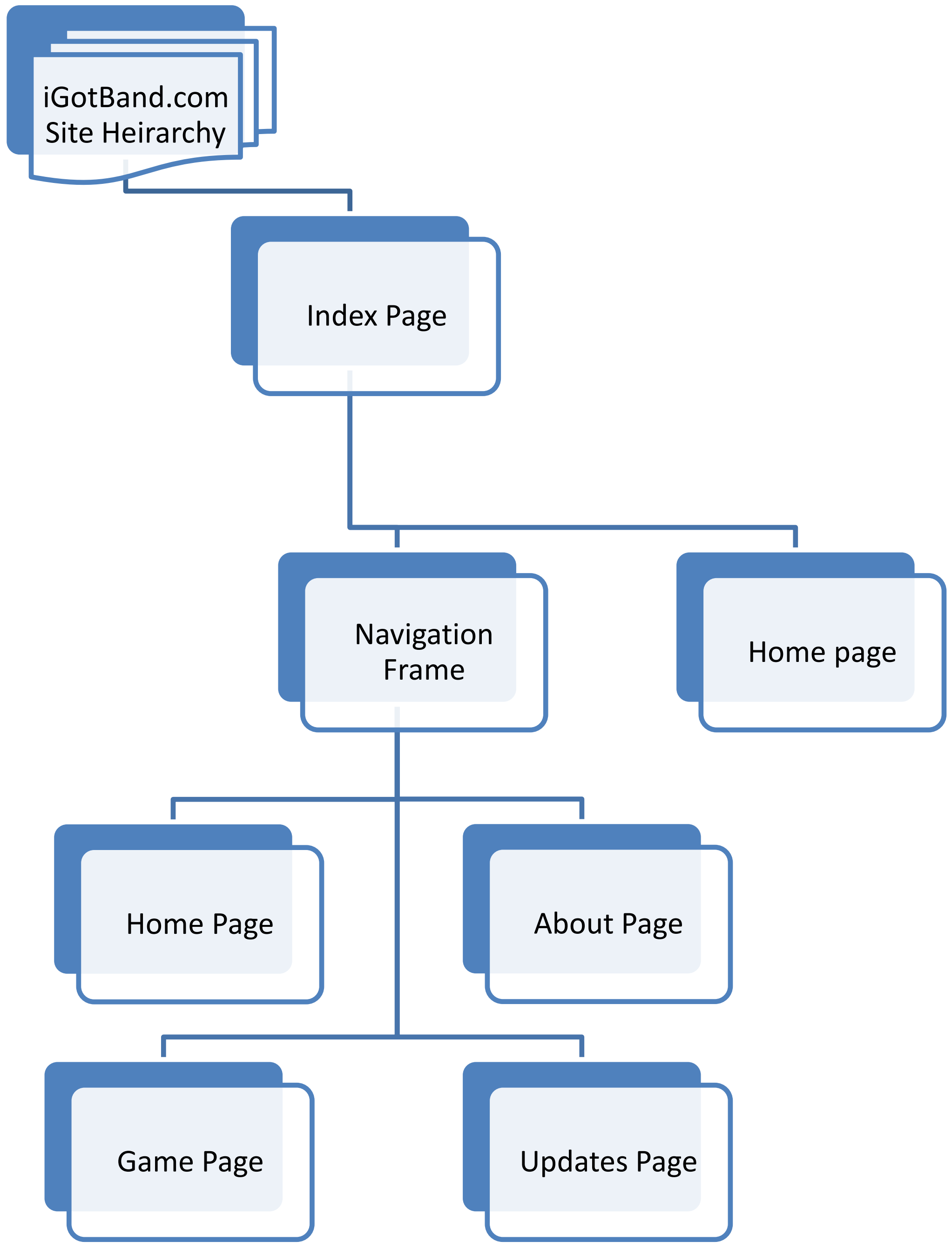
Joshua Rosenstock - Advisor

Brian Hettrick – Lead Programmer,
Lead Sound Design
WPI '09, IMDG ART
Tim Cushman – Lead Artist
WPI '09, IMGD ART
Shelli Clifford – Lead Animator
WPI '09, IMGD ART
Alex Schwartz – Lead Character Design
WPI '09, IMGD ART

The Controls:

iGotBand can be played with any standard computer keyboard, or Xbox Guitar Controller with USB input (see: Setting Up Guitar Controller). To play with the keyboard, press and hold any of the numbers 1-5 to select the note you wish to play, and finally press the spacebar t strum the selected note.

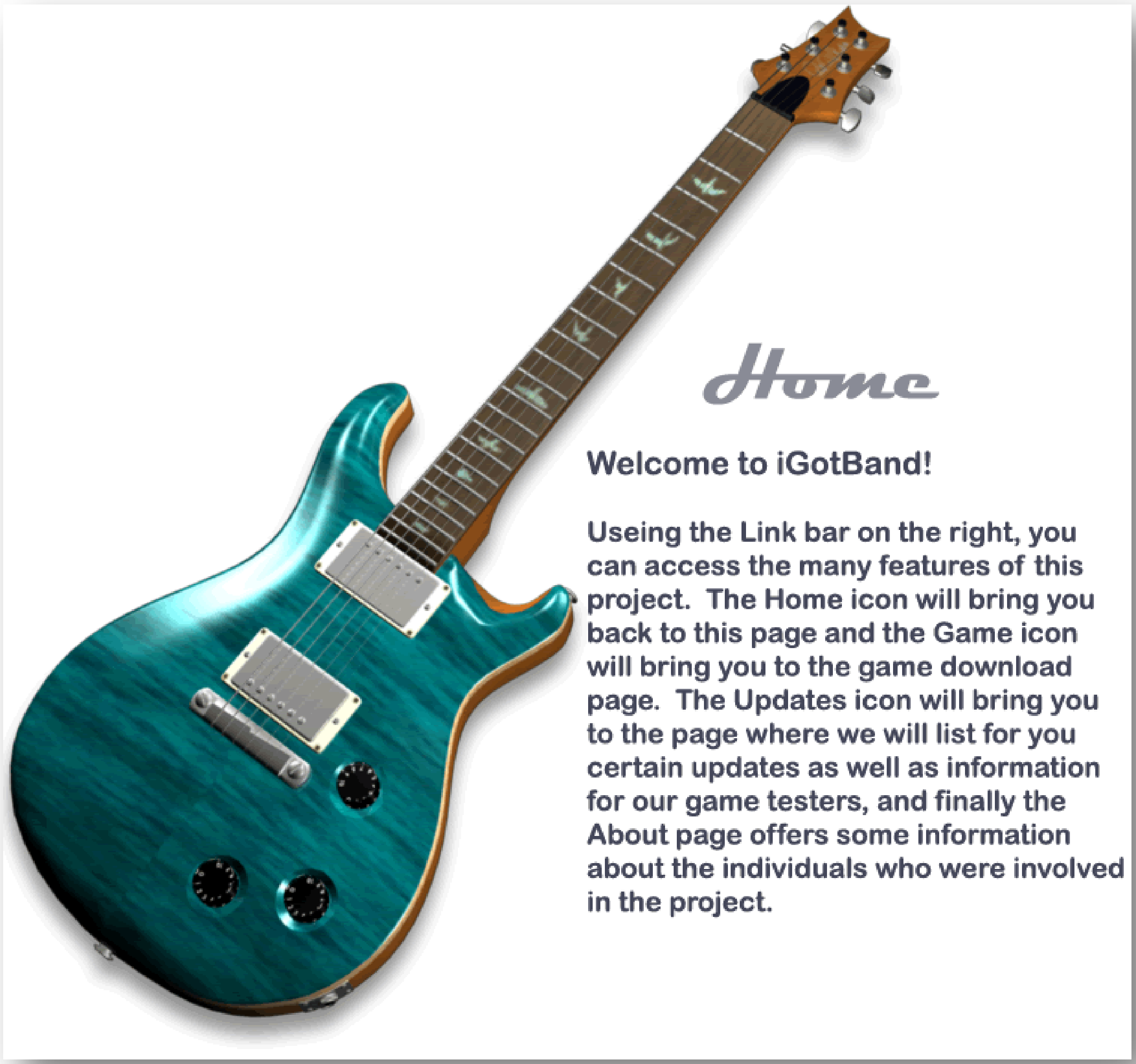
Figure 4.5:3 Final design of About page


iGotBand.com Site Heirarchy
Index Page
Navigation Frame
Home page
Home Page
About Page
Game Page
Updates Page

Code for the main frame over the code for the about page:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Untitled Document</title>
</head>
<frameset rows="*" cols="*,227" framespacing="0" frameborder="no" border="0">
  <frame src="MainPage.html" name="mainFrame" id="mainFrame" />
  <frame src="LinkNeck.html" name="rightFrame" scrolling="No" noresize="noresize" id="rightFrame" />
</frameset>
<noframes><body>
</body></noframes>
</html>
```

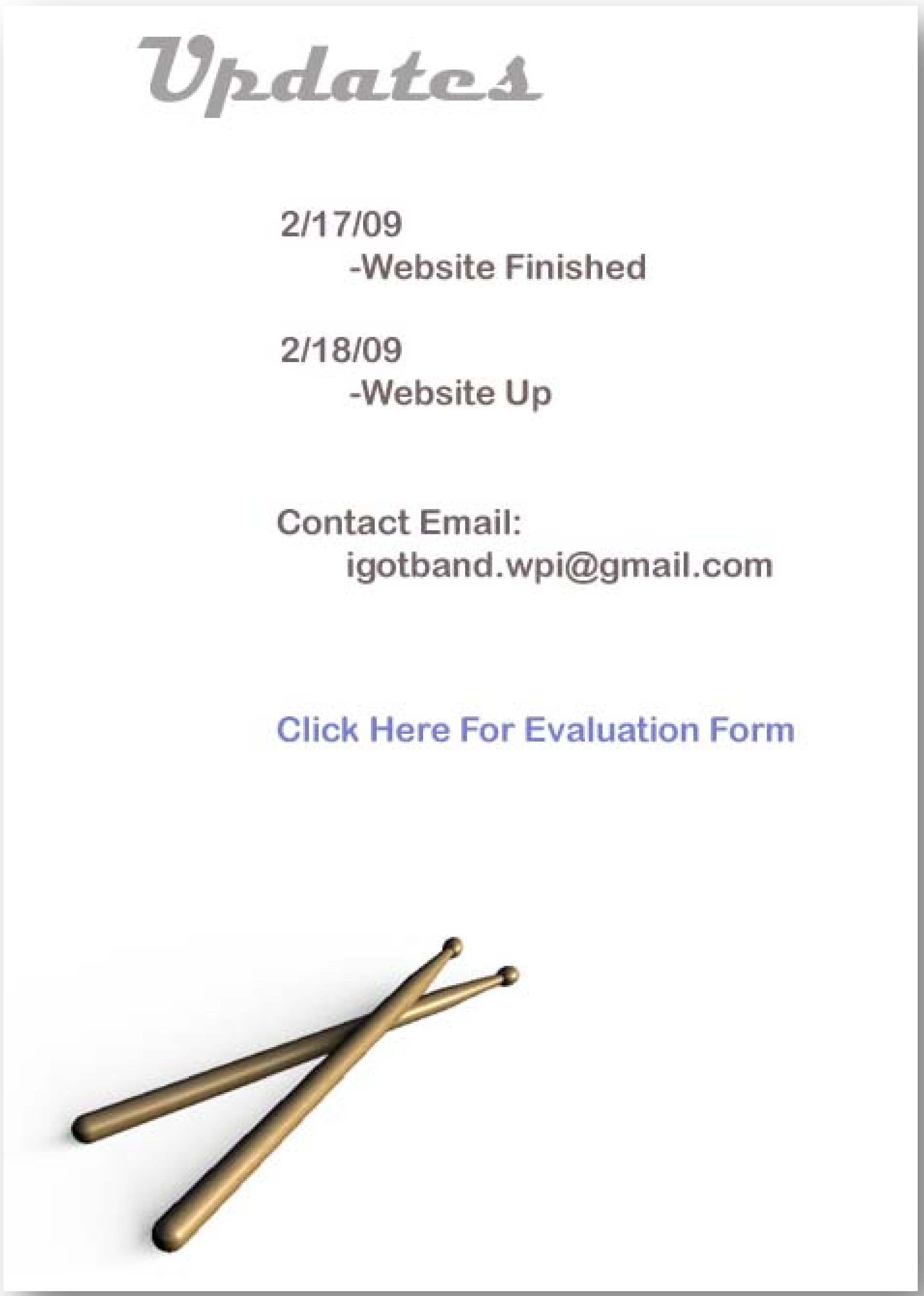
```
<html>
<head>
<title>AboutPage</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF" leftmargin="0" topmargin="0" marginwidth="0" marginheight="0">
<!-- ImageReady Slices (AboutPage.psd) -->
<table id="Table_01" width="1024" height="768" border="0" cellpadding="0" cellspacing="0">
	<tr>
		<td rowspan="2">
			<img src="images/AboutPage_01.jpg" width="503" height="768" alt=""></td>
		<td colspan="2">
			<img src="images/AboutPage_02.jpg" width="521" height="461" alt=""></td>
	</tr>
	<tr>
		<td>
			<img src="images/AboutPage_03.jpg" width="58" height="307" alt=""></td>
		<td>
			<img src="images/AboutPage_04.jpg" width="463" height="307" alt=""></td>
	</tr>
</table>
<!-- End ImageReady Slices -->
</body>
</html>
```


Home

Welcome to iGotBand!

Useing the Link bar on the right, you can access the many features of this project. The Home icon will bring you back to this page and the Game icon will bring you to the game download page. The Updates icon will bring you to the page where we will list for you certain updates as well as information for our game testers, and finally the About page offers some information about the individuals who were involved in the project.

Figure 4.5:4 Final design of the Home page



Updates

2/17/09
-Website Finished

2/18/09
-Website Up

Contact Email:
igotband.wpi@gmail.com

Click Here For Evaluation Form

Figure 4.5:5 Final design of the Update page

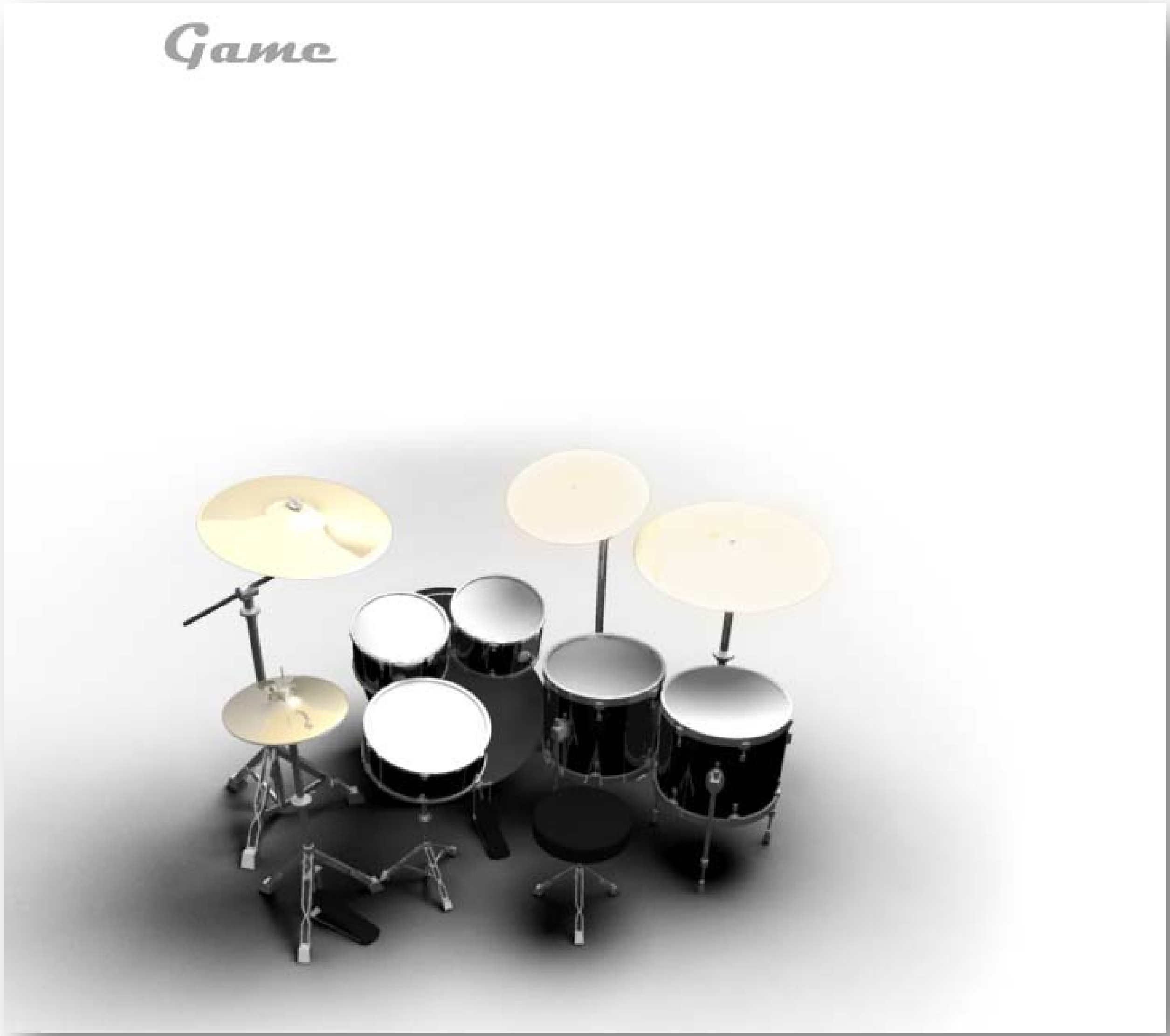


Figure 4.5:6 Final design of Game page

5. Conclusions

5.1.Post Mortem

There were no blaring failures that occurred in the course of the project, but we did fail to put in many of the 'extras' that we originally planned on adding in, such as crowd surfing fans, people who come out with t-shirt guns and other special effects when players would all play well. These, however, were never considered to be a necessity, especially considering the early problems we had with laggy graphics caused by having too many art assets on screen at once.

These extra sprites and effects that were left out have the capability to be added in the future. Another game element that is in the works for future development is the addition of a song conclusion screen. This screen would include game data of the song that was just played, such as the number of notes hit and the number of fans collected.

5.2 Late Additions and Successes

One success brought about in the game came in the realm of the note brackets that fans require. Allowing fans to ask for multiple notes played in sequence offer another level of gameplay over the original mechanic presented in our first design implementation of "one note to a fan". This previous single note bracket did not offer an accurate representation of how the player is playing compared to the number of fans they have.

Other notable successes involve elements of the game that came later in its development. One such addition was the functionality to easily make songs in the form of a text file, and add background music in the form of an .mp3 file, both of which can even be altered or added by the players. This new aspect of the game would allow a greater ease of expansion of content on the part of both the production team and those who wish to play the game.

Some other things that we added to the game for a few final touches really helped fill the game out. These aspects include the addition of music provided by Brian, a fully functioning menu interface to replace the need for the website, and an installer for both Mac and PC computer systems. The music provided by Brian were compliments of his band, Papersky, to avoid any copyright issues that may have presented themselves in the future. The menu interface includes a band, venue, and song selection that really completes the game in a professional manner. Finally the installers made by Alex vastly increase the ease in which people can interface their guitar controller and the game by automating the entire process.

5.3 Conclusions

iGotBand meets our goals as being a music game that encourages musical improvisation, is easy to access and start playing, and has potential for endless possibilities of musical experience. Some graphical features we had originally thought about were not included. But new features, including a system of creating custom music for use in game levels, a fully functional game menu interface, and game installers, that were implemented as the game was created proved to be great additions that really filled it out and made it look professional.

6. APPENDICES

6.1 RESEARCH

During the first week of our project, we researched different forms of improvisational music as well as the history behind improvisation. The sections below are summaries of the topics that we studied in preparation for our project.

6.1.1. Indeterminacy

In music, *indeterminacy* usually refers to the musical movement that rose around John Cage and his works. In his pieces, Cage used a number of devices to ensure randomness and thus eliminate any element of personal taste the performer may put into the piece (Britannica). Many of Cage's early works using indeterminacy used several methods (I Ching, rolling dice) to ensure a random collection of notes, but the pieces after the composition were fixed and were always to be played in the originally produced manner. Later works involved segments of music with notes determined by chance that the performer could choose to play certain pages of or play all at once; these works would change from performance to performance (Newgrove). Some see indeterminacy as a way to do away with the fixed properties of music, and eliminate the control a composer has over the piece created. In a most radical view, all sounds in a piece have equal value: sounds chosen by the composer, by the performer, and all the unpredictable sounds that we hear every day. Indeterminacy in this radical view is thus philosophically opposed to aleatoric music. In aleatoric music there is still an element of control present by the composer. Usually this control is in offering the performers a limited number of possibilities to choose from.

6.1.2. Aleatoric Music

The word *aleatory* comes from the Latin *alea*, “dice”. It is a style of music, mostly coming about in the 20th century, in which the realization of the composition in some way is left to chance or indeterminate methods used by the performer. “Aleatoric” can be used to describe compositions with strictly marked areas for improvisation with specific directions, and it can also refer to unstructured pieces with vague directions (Britannica gives an example of “Play for five minutes”). The indeterminate aspect of aleatory music usually occurs in two ways. The performers of the piece may be told to arrange the structure of the piece and how it is played. For example, they may reorder the sections of the piece, or they may play multiple sections simultaneously. The musical score may also indicate points where performers are told to improvise (Britannica). An early form of aleatoric music is the *Musikalisches Würfelspiel* or “musical dice game”, which was popular in the late 18th and early 19th century. These games consisted of a sequence of musical measures. Each measure had several possible versions. The precise sequence of measures to be played was determined by throwing of a number of dice. One musical dice game of this kind is attributed to Wolfgang Amadeus Mozart (Boehmer 1967, 9–47).

6.1.3. Extended Technique

Extended techniques are unconventional, unorthodox, experimental, or “improper” techniques of singing or of playing musical instruments. The use of extended techniques is common among jazz musicians and music in free improvisation. Popular music also makes use of extended techniques. Examples of extended technique:

Beatboxing

Prepared guitars or pianos, where objects have been placed on or under the strings to produce new effects and sounds

The use of unusual bowing techniques with string instruments. Examples include *sul tasto* (using the bow over the fingerboard), and *col legno* (hitting the strings with the wood of the bow).

Using a mouthpiece on one instrument with the body of a different instrument.

Turntablism, the use of turntables to turn records and mix sounds

In brass instruments, flutter-tonguing, growling or humming while playing, or using an unusual mute.

6.1.4. Graphical Notation

Graphic notation is a form of music notation which refers to the use of non-traditional and non-standard symbols and text to convey information about a piece of music and how it is to be performed. It is used for experimental music; in many cases it is difficult to notate experimental music using standard notation. There are many forms of graphic notation:

Prose scores: Music and its directions are written as ordinary text, and the interpretation is left to the performer.

Graphic scores: Music is represented using symbols and illustrations, usually blocks of color

Piano Roll Notation: Music is displayed as blocks in line with the notes on a piano. This notation is now popularly used in music composing software to create music. Examples of piano roll notation can be found in FruityLoops, Reason, and OrgMaker

In other forms, music is shown as line staves, showing a relative pitch, with the actual pitches being decided upon performance

Specific graphic symbols also appear in avant-garde and experimental works, giving more specific directions (as opposed to general graphic scores giving way to many different interpretations of a piece).

6.1.5. The Oblique Strategies

The Oblique Strategies is a deck of cards made by Brian Enos and Peter Schmidt. They developed the idea of the cards through thinking about approaches to their own work as artist and musician. On each card is printed a vague phrase, coined a “worthwhile dilemma” by the makers. Each is a suggestion of a course of action to take or a way of thinking to assist one in creative situations. Three editions were made in limited quantity. A fourth edition was privately commissioned after Peter Schmidt died on holiday in 1980. Despite all claims that no further editions plan to be made, there is apparently a “fifth” edition available. There are also versions in French and Japanese, and a fan made set of contributions called “The Acute Strategies”.

6.1.6. Game Pieces

Game pieces refer to the series of improvised works created by John Zorn. These compositions involve a group of musicians who follow a series of rules and cue cards. Strategy is also an important part of a game piece because each player has the opportunity to change the direction of the music. There are no prerecorded sequences or riffs, so the outcomes are always unique.

The rules for game pieces are somewhat complex. First, the prompter selects a player (players may raise their hands to attract attention). The chosen player will then signal the prompter by pointing to a part of his or her body and by holding up fingers. The prompter then holds up a cue card that matches what the player has signaled, and the rest of the band must perform according to the rule corresponding to that cue card. This process is repeated until a player signals an ending cue. In a game piece, there are several types of actions, or "classes" of cues, for the players to choose from. The body part signaled by the player corresponds to one of these classes, which may include musical direction such as change of ensemble, solo and duo improvisation, volume changes, and endings. There are also "guerrilla systems" which involve tactics that may disrupt the game, making the outcome more interesting. Although the players are aware of the rules of each game piece, Zorn never tells the listeners what the rules are. Therefore, each performance yields unexpected outcomes, especially for the listeners. While some of the rules may become apparent during a game piece, there is always a sense of excitement and uncertainty. Listening to a game piece has been described as "listening to controlled chaos". There are several different versions of game pieces, often named after popular sports games such as Baseball, Lacrosse, Curling, Golf, Hockey, Cricket, Fencing, Pool, and Archery. Perhaps the most famous of such works is Cobra, which was first recorded and released in 1987 and is still being performed today.

6.1.7. An Anthology of Chance Operations

An Anthology of Chance Operations is a collection of different forms and styles of randomness that is incorporated into an art form. Although it covers different fields such as literature and poems, there is a good amount of information on music that is covered.

The random distribution of variations on a set performance to be interpreted by the performer

The random orientation of segments of a piece, whose orientation is determined by the performer

The random physical location of the performer while playing a piece

The random distribution of instructions that are performed at a changing physical orientation to be determined by the performer

The giving of random instructions to a performer to be played

The use of repetition

The use of Free-Form improvisation

6.1.8. Computer games that use real instruments

Guitar Rising

Guitar Wizard

Piano Wizard

The Improviser: a Musical Improvisation Game

Rock Band with a V-drum drum kit

A real bass pedal with rock band

6.1.9. References

Hitchcock, H. Wiley. “Cage, John (Milton, Jr).” The Newgrove Encyclopedia of American Music. Stanley Sadie. Vol. 1. London: Macmillan Publishers Limited, 1986. “Indeterminacy in Music”, Wikipedia. <http://en.wikipedia.org/wiki/Indeterminacy_in_music>. “Aleatoric Music”, Encyclopædia Britannica. 2008. Encyclopædia Britannica Online. <http://www.britannica.com/EBchecked/topic/13676/aleatory-music >. Boehmer, Konrad. 1967. ''Zur Theorie der offenen Form in der neuen Musik''. Darmstadt: Edition Tonos. (Second printing 1988.) “Extended Technique”, Wikipedia. <http://en.wikipedia.org/wiki/Extended_technique>. “Graphic Notation”, Wikipedia. <http://en.wikipedia.org/wiki/Graphic_notation>. “The Oblique Strategies”. <http://www.rtqe.net/ObliqueStrategies/index.html>. “John Zorn Biography”. <http://www.scottmaykrantz.com/zorn01.html>. “John Zorn”, Wikipedia. <http://en.wikipedia.org/wiki/John_zorn>. “Cobra”. <http://www.bookrags.com/wiki/Cobra_(Zorn)>.

6.2.Other Art Assets

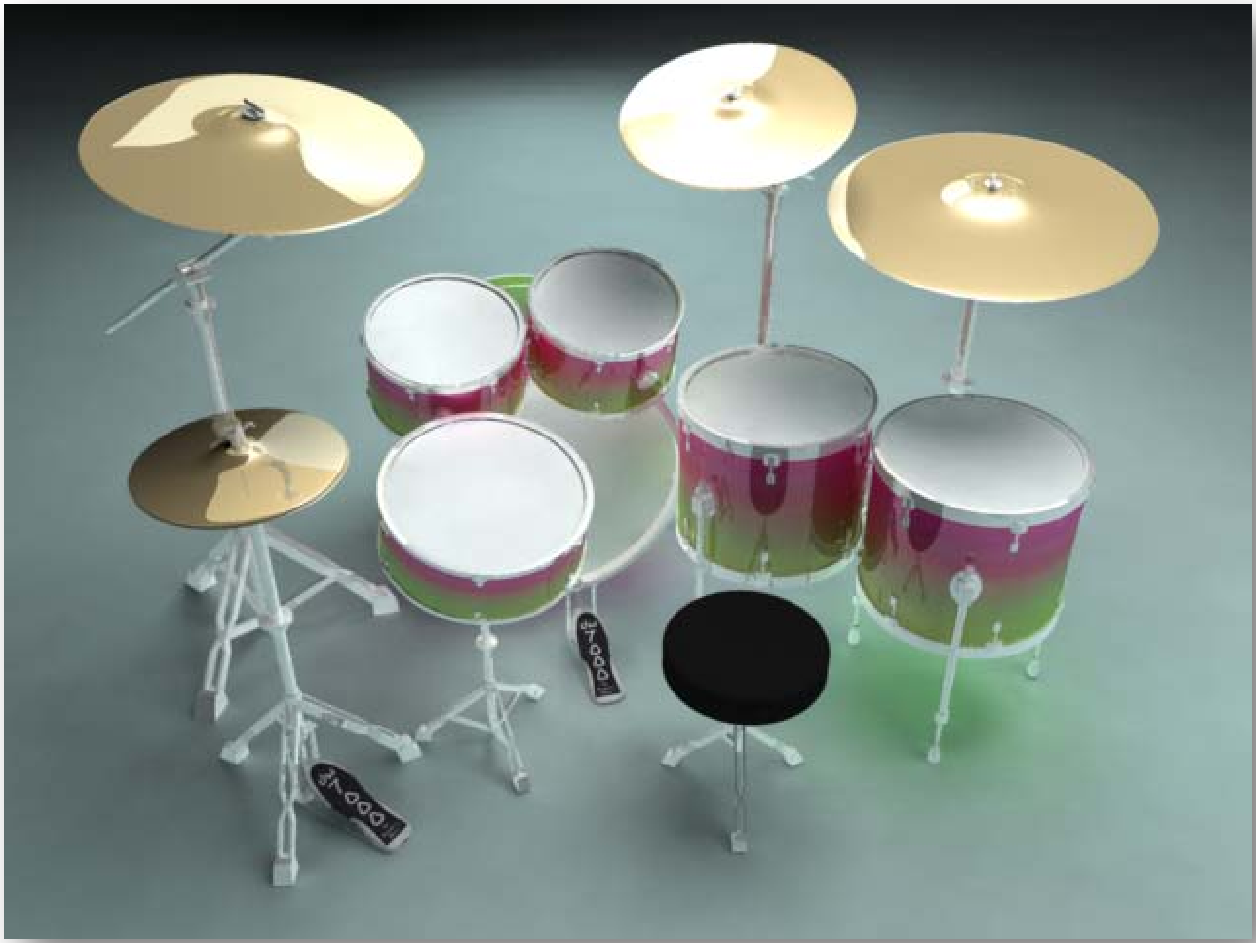


Figure 6.2:1 Drum set, Modeled and Textured by Alex

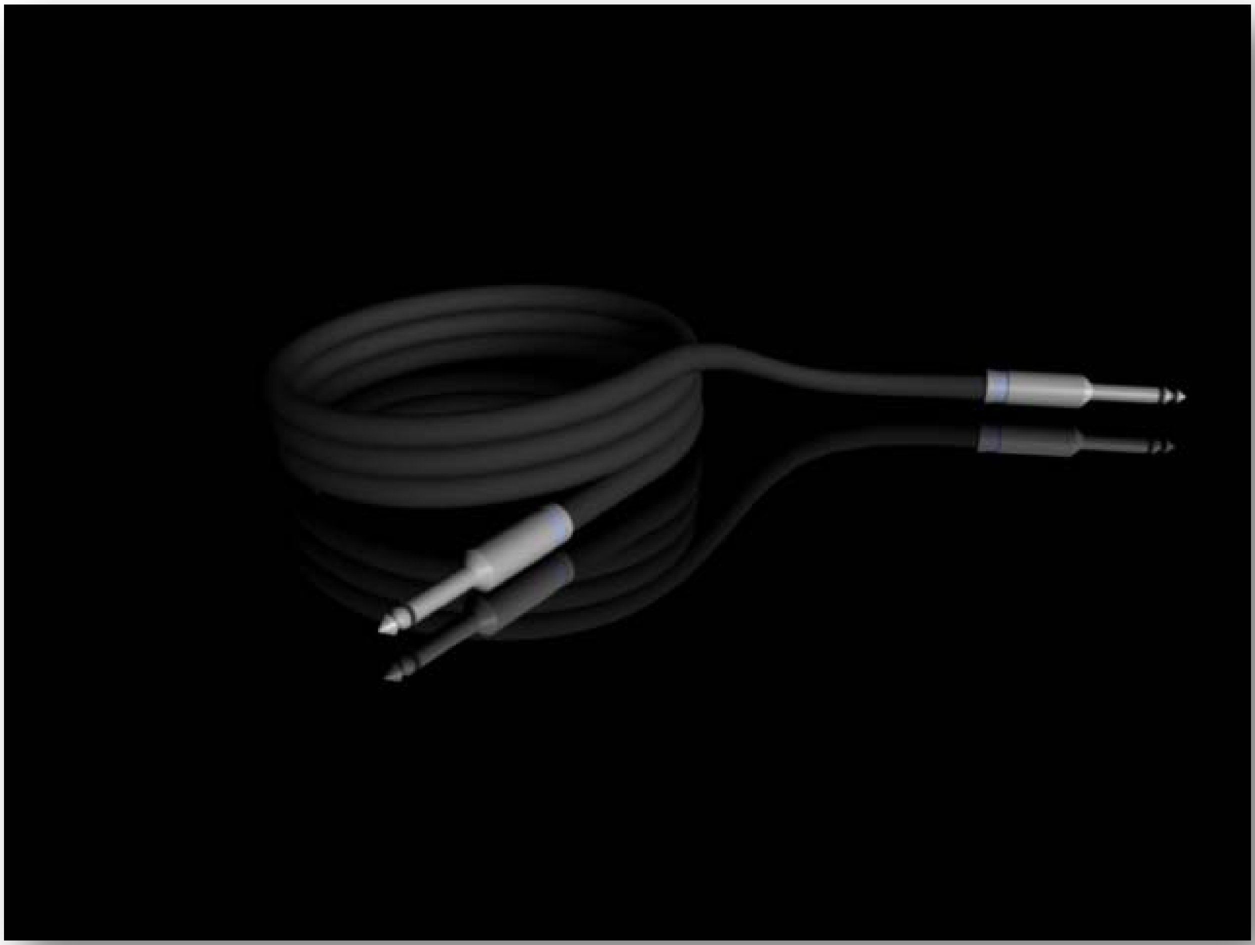
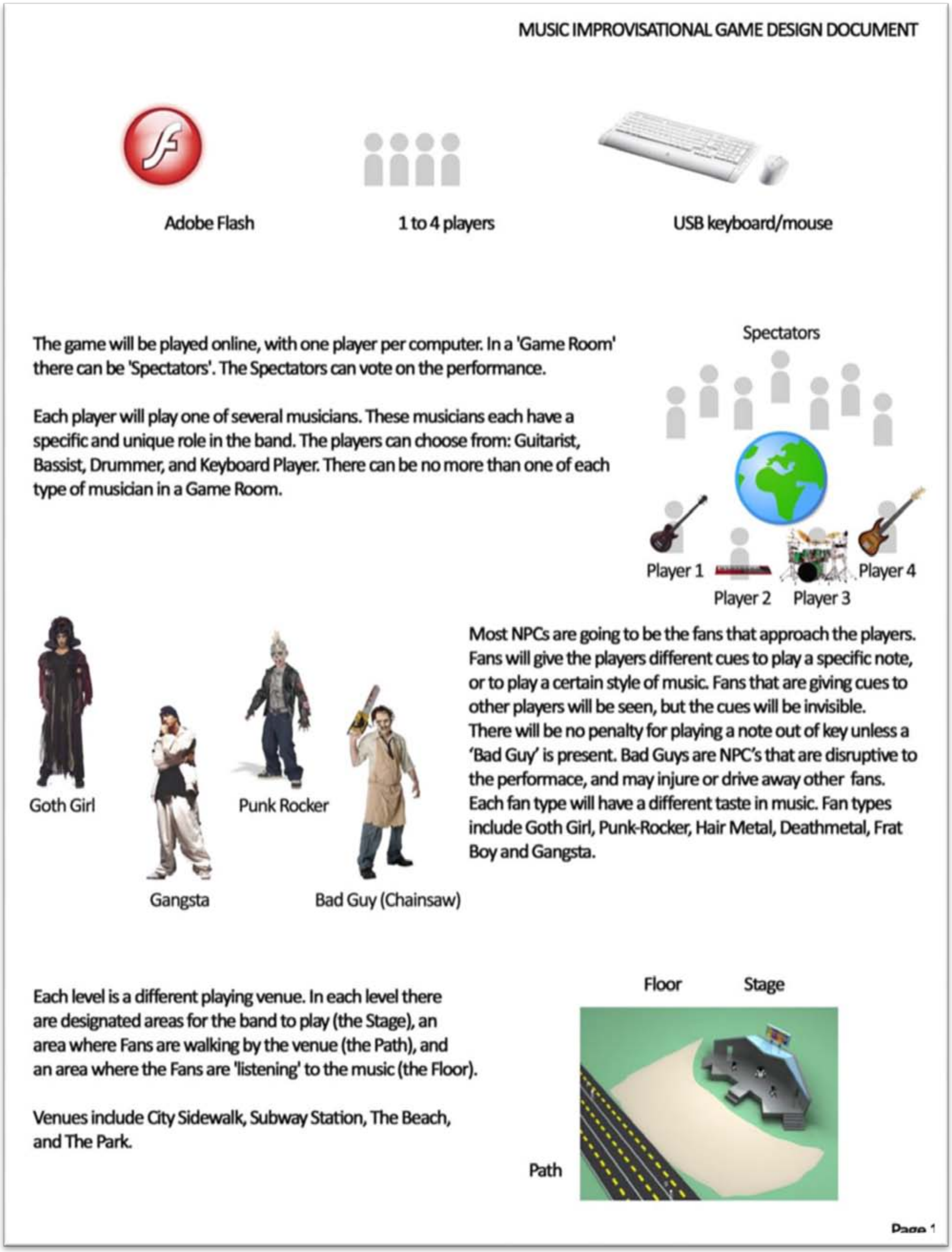


Figure 6.2:2 1/4in cable, Modeled and Textured by Tim

6.1.Original Idea Game Doc



MUSIC IMPROVISATIONAL GAME DESIGN DOCUMENT

Adobe Flash

1 to 4 players

USB keyboard/mouse

The game will be played online, with one player per computer. In a 'Game Room' there can be 'Spectators'. The Spectators can vote on the performance.

Each player will play one of several musicians. These musicians each have a specific and unique role in the band. The players can choose from: Guitarist, Bassist, Drummer, and Keyboard Player. There can be no more than one of each type of musician in a Game Room.

Spectators

Player 1

Player 2

Player 3

Player 4

Goth Girl

Gangsta

Punk Rocker

Bad Guy (Chainsaw)

Most NPCs are going to be the fans that approach the players. Fans will give the players different cues to play a specific note, or to play a certain style of music. Fans that are giving cues to other players will be seen, but the cues will be invisible. There will be no penalty for playing a note out of key unless a 'Bad Guy' is present. Bad Guys are NPC's that are disruptive to the performace, and may injure or drive away other fans. Each fan type will have a different taste in music. Fan types include Goth Girl, Punk-Rocker, Hair Metal, Deathmetal, Frat Boy and Gangsta.

Each level is a different playing venue. In each level there are designated areas for the band to play (the Stage), an area where Fans are walking by the venue (the Path), and an area where the Fans are 'listening' to the music (the Floor).

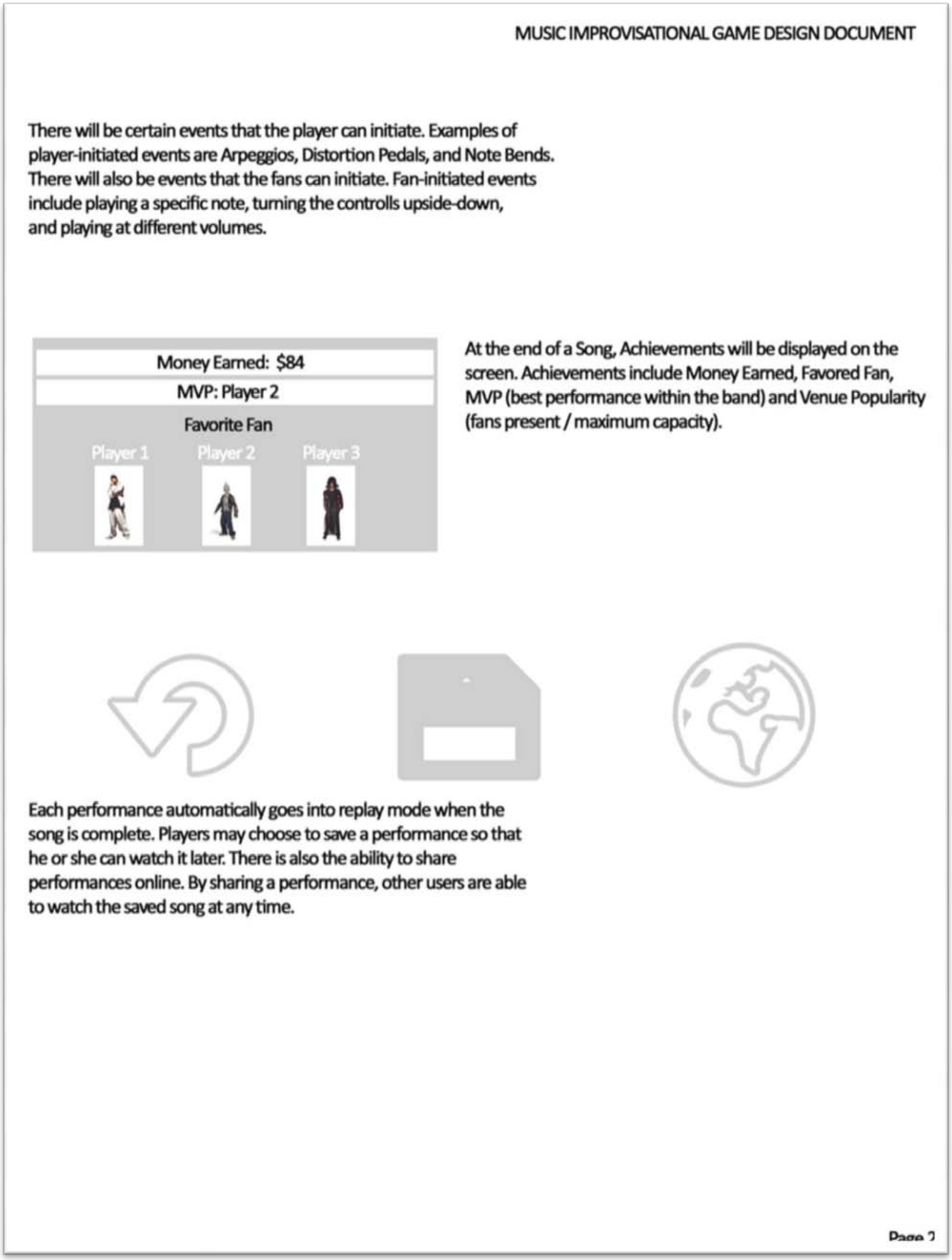
Venues include City Sidewalk, Subway Station, The Beach, and The Park.

Floor

Stage

Path

Page 1



There will be certain events that the player can initiate. Examples of player-initiated events are Arpeggios, Distortion Pedals, and Note Bends. There will also be events that the fans can initiate. Fan-initiated events include playing a specific note, turning the controlls upside-down, and playing at different volumes.

Money Earned: $84
MVP: Player 2
Favorite Fan
Player 1
Player 2
Player 3

At the end of a Song, Achievements will be displayed on the screen. Achievements include Money Earned, Favored Fan, MVP (best performance within the band) and Venue Popularity (fans present / maximum capacity).

Each performance automatically goes into replay mode when the song is complete. Players may choose to save a performance so that he or she can watch it later. There is also the ability to share performances online. By sharing a performance, other users are able to watch the saved song at any time.

6.3.Improv Online Game Doc

IMPROVISATIONAL MUSIC GAME

DETAILED DESIGN DOCUMENT

Version 2.0

October 16, 2008

Shelli Clifford, Tim Cushman,
Brian Hettrick, and Alex Schwartz

CONTENTS

1. PREFACE

In this project, we wish to create an online community where bands and other performers can interact with fans around the world in a friendly and constructive environment. The game will encourage performers to improvise and interact with the audience, as well as encourage fans to interact with the game as much as possible. Ultimately we would like to allow people to experience a wide range of improvisational music through this game.

1.1 Purpose

The purpose of our game is to explore new grounds in rhythm and music games, a culture that has become overwhelmingly popular over the last decade. We have decided to approach this style of gaming from a new angle by creating an open-ended music game that relies heavily on community interaction.

1.2 Target Audience

The target audience is dependent on the music being performed. The general age range of the target audience is from young teens to young adults. The game specifically appeals to those who are genuinely interested in music and the performing arts, and those who enjoy live interactions with other people.

1.3 Prerequisites

Since the game will be published in Flash, any system running the application must have the latest version of Flash Player installed. In addition, the game deals with streaming audio, so players must have a stable connection to the Internet. For performers, an input device and compatible sound card is also required. Examples of input devices are USB microphones, mini-plug microphones and midi keyboards that are synced to software synthesizers.

2. DESIGN CONSIDERATIONS

2.1 Assumptions, Dependencies and Risks

We assume that Actionscript 3 will allow streaming audio inputs and outputs to multiple clients. We also assume that Flash will be able to render hundreds of bitmaps on one screen at the same time. If rendering becomes an issue, we will need to flatten the pieces of each avatar into single characters and save each user-created combination on our server. By saving the combined avatars, we reduce the number of bitmap instances that need to be rendered over each other on a single screen.

Our game will be dependent on the capabilities of our server. The server must support PHP and streaming audio, and must be able to handle a large number of users accessing the same information at once. We are also depending on user interest. Ideally, our users will prefer listening to the live stream of a band over a pre-recorded session of the same music. However, we must also consider the sound quality of streaming audio. CD's playback at 320 kilobits per second, and most mp3's playback at a bit rate between 128 and 192 kilobits per second. Streaming quality will be dependent on the bandwidth of the clients' internet connections. Since the fidelity of streaming audio will most likely be less than that of its pre-recorded counterparts, there is a need for a novelty aspect to our system.

Since we will be hosting personal information such as email addresses, we run the risk of having hackers obtain that information through our server. Personal information, as well as vital game data, must be encrypted and protected properly to minimize that risk.

2.2 General Constraints

Visually, we are constrained to the graphics capabilities of Flash. Therefore, most of the sprites and 3D sequences must be pre-rendered. In addition, the game will be playable through a browser window, so screen size is also limited to the viewable area of the monitor minus the browser's interface.

Since the game will be distributed online, size of the game file is also constrained. An alternative is to allow users to download the client so that the static information can be accessed locally.

Another constraint is gameplay. Since gameplay is purely online, players will not be able to run our game without a stable connection to the Internet. There will not be a single-player mode because login requires access to our server.

2.3 Goals and Guidelines

Ultimately, our goal is to create a friendly environment that allows our users to play shows and get constructive feedback from the fans that listen. Our game should also encourage performers to improvise freely and be rewarded for good improvisation. The rewards and items should act as incentives for getting more involved and interacting with other players, and should not be abused. In our design we will constantly test situations in which players might attempt to abuse the system, and we will minimize the possibility of such events.

In order to have a functional game we must have:

- A new account sign-up interface for new users
- PHP script that accepts a new sign-up and sends a confirmation email with a unique link
- PHP script that confirms a user's email account when the link is followed
- PHP script that creates a new folder and default data files for new users
- Actionscript that allows users to login and edit their information in game by communicating with server-side PHP script
- Actionscript that enables I/O streaming media through Flash
- Actionscript that enables user avatars to roam the virtual world and interact with the interface

using key strokes and mouse clicks

- A rating system that functions in Flash and communicates with server-side PHP script to rewrite information in user folders.
- Images for each element of the user interface
- 3D models for basic male and female character
- Textures for 5 types of outfits for both male and female characters
- 3D models and textures for a guitar, bass guitar, drum set, keyboard, microphone, microphone stand, and instrument amplifiers
- 3D models and textures for a stage and the surrounding environment
- Rendered images of character models, instrument models and environment model, imported as animations into Flash
- Sound effects and ambient noise

2.4 Development Methods

All 3D models will be created in Autodesk Maya 2008. Since the actual models will not be imported into the game, the meshes do not have to be perfect; there may be missing sides or faces as long as they do not show when rendered. Textures will mostly be created in Adobe Photoshop. Real photographs may also be used to create textures. Once the textures are applied, each model will be rendered in a common room, so that lighting and render properties are exactly the same. The rendered images will be saved as PNG files and imported into Flash.

All of the back-end mechanics will be programmed in PHP. PHP scripts will be used to create, edit, delete, and relay information from our database to the clients' Flash documents. Flash will then display the information using Actionscript. Similarly, any changes made in the game will be sent to a PHP script, which will rewrite information in our database.

PHP and Actionscript will also be used to stream audio. The scripts must read in audio from a performer's computer and output the same audio to all of the listeners' computers. The input and output of audio will be done in Actionscript, and server-side relaying will be done through PHP scripts.

Sound effects and ambient noise will be recorded with an AKG Perception 200 microphone and Digidesign Mbox 2. The recorded audio will be edited in Digidesign Protools and saved as mp3 files to be imported into Flash.

3. DATA DESIGN

On the server there will be a SAFE folder, located above the public site directory so that it cannot be accessed directly by public users. In this SAFE folder there are 2 folders: users and bands. In the users folder there are individual folders for each user account. Each individual user folder contains login information, account information and player data. The bands folder contains information about each band that has been created. The most important pieces of information in this folder are band member usernames, reputation points and logo.

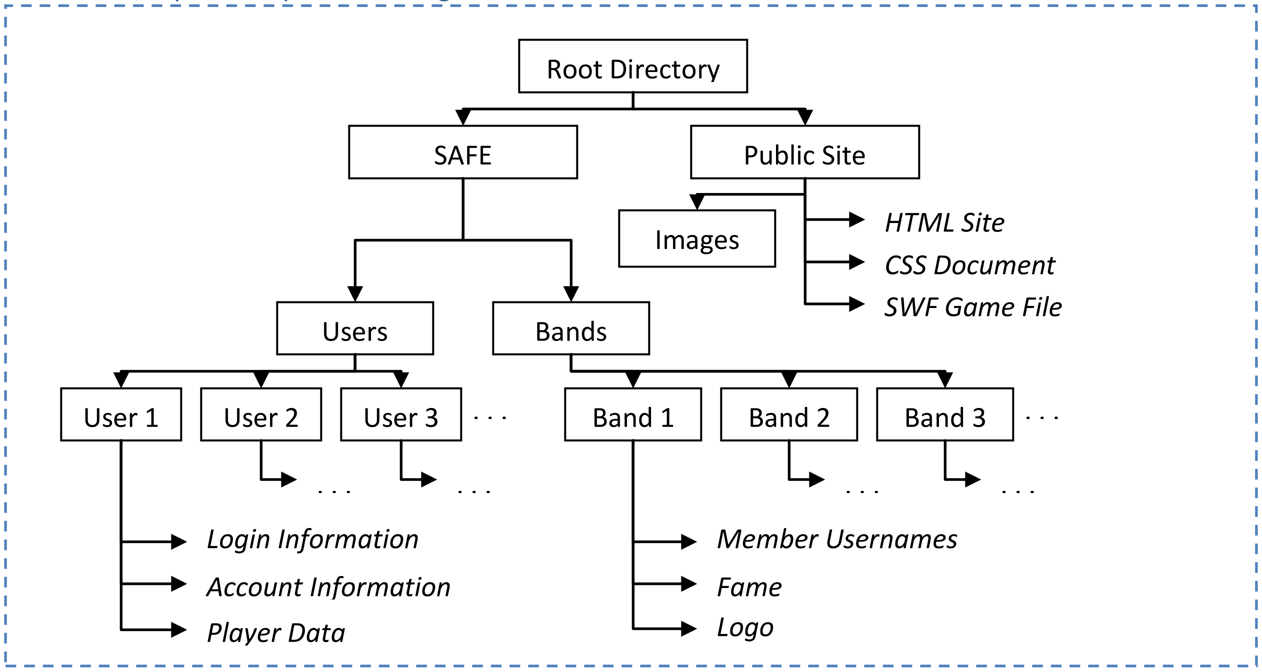


Figure 3.1 Server database structure

3.1 User Database

In Figure 3.1, the user account folders (User 1, User 2, User 3, etc.) will be replaced by usernames. A username is a unique identification name for an account, which is chosen by the user during account creation. There will never be duplicate usernames, regardless of character capitalization. A Username may not be changed once an account is created.

Login information includes username and their encrypted password. A password, created when a user signs up for a new account, will be encrypted by inputting the password into a standard Unix DES-based encryption algorithm. Usernames will not be encrypted.

Account information contains data such as their email address and the date the account was created. As far as the game engine is concerned, this information is read-only, and will generally not be accessible to other users in the game.

Player information includes the user's specific avatar combination, and the amount of in-game points and items he or she currently has. An avatar combination is simply an array that determines which bitmaps images to render for certain parts of the avatar. Shirt, pants, hair and skin color information will be stored in the avatar combination array.

3.2 Band Database

Similarly, the band folders (Band 1, Band 2, Band 3, etc.) will be replaced by band names. Band names are also unique identification names for bands. However, unlike usernames, band names may be changed by its members.

The *member usernames* file contains a list of the usernames of the members of a band. A *member usernames* file must contain at least one username. Usernames can be added or deleted from the list at any time by the creator of the band.

Fame points are earned by a band when playing shows and receiving positive feedback from its listeners. The *fame* file contains a number, specifying the number of points the band currently has.

The *logo* file is simply a PNG image of the band's logo, which can be uploaded and changed by members of the band. A logo is used to brand a certain band's merchandise.

4. INTERFACE DESIGN

5.1 Art Assets

Character models will be based off of Unreal Tournament meshes. The meshes will be modified and textured in Maya. For each body part (head, torso, and legs) we will create the following styles of clothing:

- Default – Faded jeans and plain t-shirt
- Goth – Dark and de-saturate colors with metal accessories
- Punk – Bright 80’s colors, crazy hair and skater shorts
- Businessman – Suit, briefcase and trim clean hair
- Gangster – Jewelry, long tee shirt, baggy jeans and sneakers

For each model we will render the following animation sequences:

- Idle pose
- Walking
- Playing drums
- Playing the keyboard
- Playing guitar
- Playing bass
- Jumping
- Head-banging
- Cheering

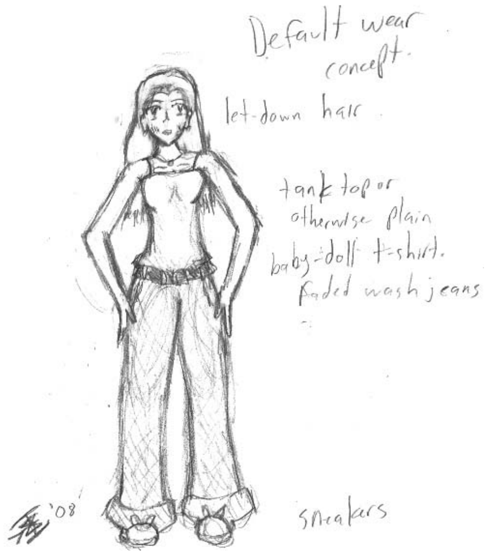


Figure 4.1 Default avatar sketch

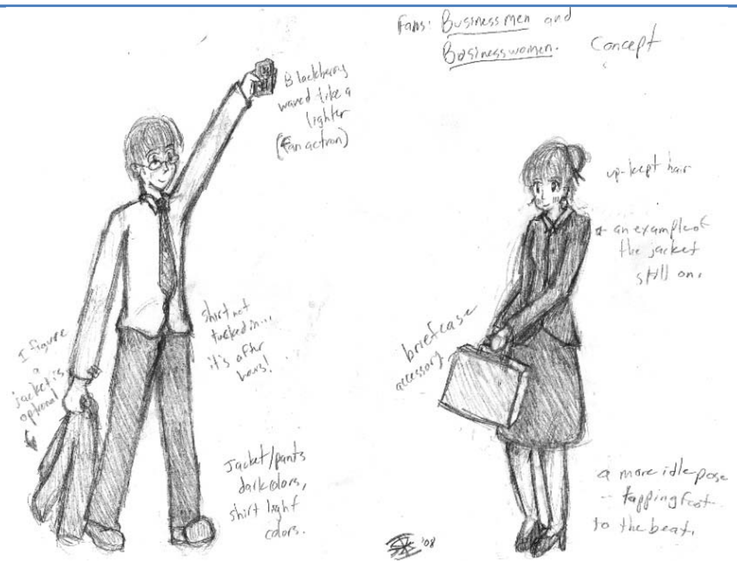

Fans: Business men and Business women. Concept
Blackberry waved like a lighter (fan action)
up-kept hair
an example of the jacket still on.
shirt not tucked in... it's after hours!
I figure a jacket is optional
briefcase accessory
Jacket/pants dark colors, shirt light colors.
a more idle pose -- tapping foot to the beat.
'08

Figure 4.1 Business avatar sketches

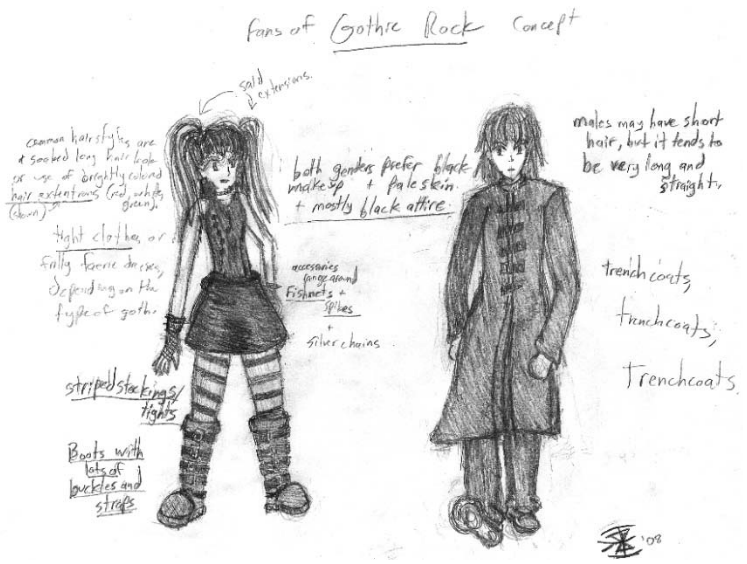

Fans of Gothic Rock Concept
said extensions.
common hairstyles are a soaked long hair look or use of brightly colored hair extentions (red, white, green). (shown)
both genders prefer black makeup + pale skin + mostly black attire.
males may have short hair, but it tends to be very long and straight.
tight clothes, or frilly faerie dresses, depending on the type of goth.
accesories range around Fishnets + spikes + silver chains
trench coats, trenchcoats, Trenchcoats
striped stockings/tights
Boots with lots of buckles and straps
'08

Figure 4.1 Gothic avatar sketches

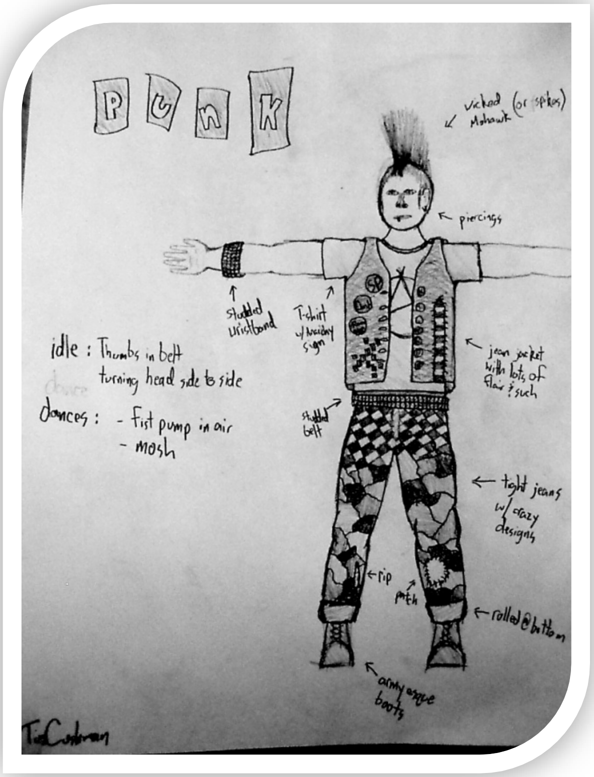

Punk
vicked Mohawk (or spikes)
piercings
studded wristband
T-shirt w/ Anarchy sign
jean jacket with lots of flair & such
idle : Thumbs in belt
turning head side to side
dances : - Fist pump in air
- mosh
studded belt
tight jeans w/ crazy designs
rip
patch
rolled @ bottom
army esque boots

Figure 4.1 Punk avatar sketch

5.2 User Interface

The game world is a “city” in which the user is free to roam. Certain locations on the map are venues for players to perform in. The list of venues includes a coffee house, park bench, and underground live house. When a user decides to enter a venue they can view the “venue lobby”, which is a list of bands that are currently performing at that venue. Users may sit in on different performances going on at that time.

Once in a room, users can hear the streaming audio from the performers, and also have the ability to chat within the room. After a certain period of time, a prompt will appear for the listeners in a room. The fans as a group must vote on the next playing style, which becomes effective in a certain amount of time after the prompt shows up. The band may choose whether or not to obey the playing style that has been voted upon. However, performers will usually gain favor of the fans by playing what they want to hear.

Users that have seen a band perform for a certain amount of time become “fans” of the band. Once a fan, a user can vote on that band (1 to 5 stars). By requiring a user to sit in on a certain amount of performance time, we can minimize the risk of users abusing this system.

Each star will add to a band’s “fame” total, and the average number of stars per user will become the band’s “quality” rating. A band needs a certain amount of fame to play in larger venues. Hence, beginner bands are limited to small venues. Both the performers and the fans that are watching a show constantly earn points by participating. With these points, users can purchase new accessories, gear, and also use them to pay for booking larger gigs.

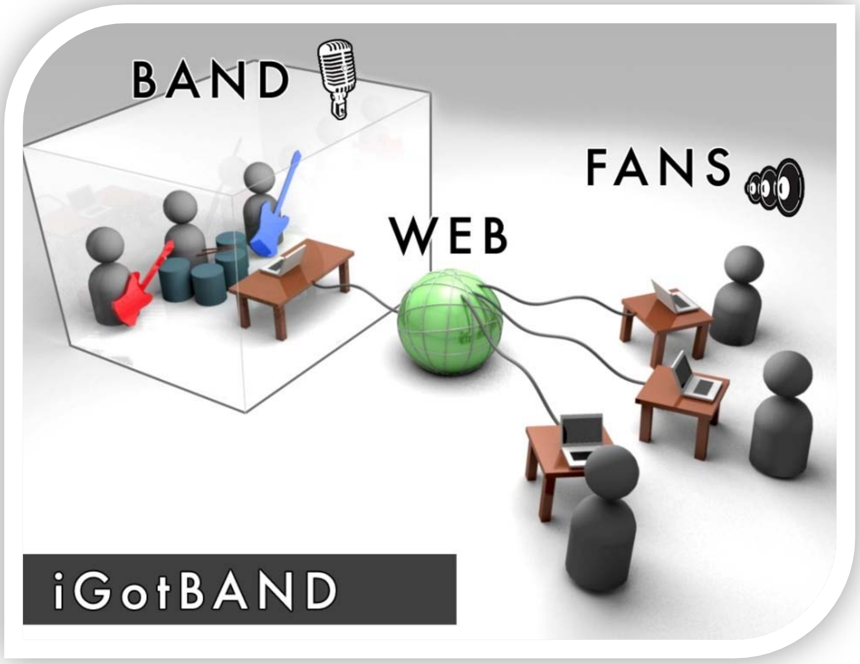
Users can also favorite certain bands and view their profiles. By adding a band to their favorites list, a use becomes a “fan” of that band. Band profiles will show their name, fame points, and number of fans they have.

To prevent people from running the program in the background of their computer rather than interact with the game, there are certain incentives for staying actively involved. At a certain interval, a random collectible, band memorabilia or gear is thrown from the stage to the audience, and the first user to grab the object gets it into his or her inventory. Such items may include drumsticks, tee shirts, picks, and broken guitar parts.

There will also be a timed “bouncer” to prevent idle crowd members. After a certain amount of inactivity, a user is “bounced” to an area directly outside of the venue. Music will continue to play, but the user will not gain any points. This method allows those who only want to listen, instead of get involved in the game, do so seamlessly. At the same time, it acts as an incentive to be part of the community and stay in the game.

5. PROCEDURAL DESIGN

Figure 5.1 Information flowchart



5.1 Server-side Scripts

All of the server-side scripts will be written in PHP. These scripts are responsible for creating, activating, editing, deleting, and outputting data. The PHP scripts will contain the code necessary for accessing information within the server that cannot be accessed by public users. By placing sensitive information above the public site directory, we can minimize the threat of hackers obtaining or hindering data on our server.

A script for creating data will be executed when someone attempts to create a new account on the site. When the PHP script receives the request, it will create a new folder and will rename this folder with the username of the new account. Within this folder it will create a *login information* file, an *account information* file, and a default *player data* file which contains initial values. *Login information* will also contain a piece of data that signifies that the account has yet to be activated, along with a secret string of digits that is needed to activate the account.

Meanwhile, the new user receives an email confirmation that directs him or her to follow a link. Within the link is a variable that has the same secret string of digits as those in his or her account's *login information.* By following this link, the PHP script for activating an account is executed. This script simply changes the piece of data that states that the account is not active, and changes it to a contrary value.

The PHP script used for editing data will be executed when a user goes into a menu within the Flash document. A user may change his or her avatar in this menu, and the script will change the avatar combination array accordingly. This script is also executed when a user gains, loses or trades points or items in the game.

When a user decides to delete his or her account, the script for deleting an account is executed. This script will prompt the user for a password. If the password inputted matches the saved password after

an encryption algorithm, the user's entire folder gets deleted.
Streaming audio must be read in by a PHP script in order to be relayed to other clients. Similarly, all text messages must also be relayed by PHP. The script will take all of the users' inputs and send them to the clients that are requesting that information.
Finally, the script used for outputting data is executed whenever a piece of data needs to be sent to the Flash document. Most information will travel between the database and the Flash document through PHP code.

5.2 Client Design

The client-side Flash application is mainly used for its interface features. Flash makes it easier to visualize data and allows users to change data without having to go through hard code. Whenever a user presses a key, the Flash document will send that information to PHP scripts in order to share the change with other users. For example, an avatar moving from one position to another must be reflected on all computers that are running the game. The Actionscript on each client allows for that to happen.
In addition, the Flash document has a role of rendering images. Each bitmap image on the server must be read by the Flash document on every client and displayed as a game world on the monitors. Flash will layer images over one another to give users the illusion that they are experiencing a real-world environment.
Lastly, the client-side application must read in text messages and streaming audio in order to send them to the PHP scripts. A "listener" in Flash will do just that, by listening to keystrokes or audio ports and sending the information when it is received. A player must calibrate his or her audio within the Flash application so that the input is being recognized by Flash. This will be done by setting the Flash "listener" to target a certain port, specifically the one that is processing the music.

6. POLICIES AND TACTICS

6.1 Design Decisions

The original plan of this project was to create an improvisational rhythm game that can be played online. The fundamental difference between the original plan and the current one is that the old idea involved computerized judgment of improvised music, and the new one depends entirely on user feedback.

We can compare specific aspects of the former design plan and the current design plan (scored from 1-5, 5 being the most desirable):

	Former design plan		Current design plan	
Opportunity to interact with others	2	Players need to follow directions to some extent, and do not have much opportunity to interact with other human players	5	Both players and fans have the opportunity to interact with one another, through text messages, voice chat, musical changes and avatar animations
Objectives	5	Offers clear game objectives	1	Objectives are unclear, “sandbox” model
Gameplay style	4	More traditional, and possibility of single-player mode	2	Gameplay is limited to interactions with other people
Musical experience	1	Musicians will need to learn a new instrument, and will be treated like beginners	5	Musicians can play their native instruments and will immediately be recognized
Input device	2	Undecided, but possibly midi keyboards, drum pads and guitar-shaped controllers	5	Anything that can send sound to the computer
Cost for players	1	Users will need to purchase equipment that they most likely do not own	4	Most musicians probably own the equipment that they need, and amateurs may perform using an inexpensive microphone
Encourages improvisation	3	To a certain extent, but does not allow for total freedom	5	Anything goes, and performances may be expanded beyond instrumental music (poetry reading, rapping, debating)
Technical ease	4	NPC coding and creating rules	1	A lot of networking and server scripts required
Innovation	3	Interesting twist on rhythm games, but still uses the same concept	5	A new type of online community that allows infinite ways of interactions between its users

Figure 6.1 Comparison chart of former and current design plans

6.2 Deadlines

- November 6, 2008 – Account creation script done
- November 13, 2008 – Working script that accepts text input and relays text messages
- November 20, 2008 – Working script that allows streaming audio between 2 computers
- December 4, 2008 – Art assets completed, website up
- December 11, 2008 – Working alpha version of the game released
- January 22, 2008 – Document all problems of alpha version
- February 12 – Release beta version of game
- February 28 – Final Release of game

6.3 Responsibilities

The creative responsibilities of each team member are as follows:

Alex Schwartz (lead technical artist) will focus on rigging, animation, website design, UV mapping, modeling accessories, and any "technical" art work.

Shelli Clifford (lead texturing artist) will focus on texturing and modeling work for the characters, and will assist in writing Flash code and creating animations.

Brian Hettrick (lead programmer) will focus on Actionscript coding in Flash, back-end PHP work, and instrument modeling.

Tim Cushman (lead modeling artist) will focus on character modeling, character texturing, UV mapping, and creating environment models.