

Operation: Stop Core

C-C-C-C-Combo Breaker!

Technical Design Document

Version 1.0 – October 15, 2004

GAM 400B

Fall 2004

Professor Mike Moore

Technical Director – Brian Eberspacher
beberspa@digipen.edu

Tech Design by:
Peter Dunshee
Brian Eberspacher
Robert Hunt III
Peter Young

© DigiPen Institute of Technology

Table of Contents

***Introduction*.....5**

Hi-Concept..... 5

Overview..... 5

Technical Goals..... 6

Minimum System Requirements..... 6

Recommended System Requirements..... 6

Risks..... 6

***Outside Tools*..... 7**

Development Environment..... 7

Version Control..... 7

Other..... 7

***Development Team*..... 8**

***Milestones*..... 9**

Engine Proof - 11/15/04..... 9

First Playable - 12/6/04..... 9

Pre-Alpha - 2/14/05..... 9

Alpha - 3/8/05..... 10

Beta - 4/4/05..... 10

Gold - 4/18/05..... 10

***Coding Guidelines*..... 11**

General Project Guidelines..... 11

Variables..... 11

***Game Mechanics*..... 12**

Game Core..... 12

Graphics Manager..... 13

***Initialization*..... 13**

***Start Game*..... 13**

***Game Loop*..... 13**

Termination..... 13
 Object Models, Character Models and Animation..... 14

File Formats..... 14

Internal Format..... 15
 Textures and The Texture Manager..... 17
 Camera Class..... 18
 Text..... 18
 Lighting..... 18
 Sound Manager..... 19

Initialization..... 19

Start Game..... 19

Game Loop..... 19

Termination..... 20
 Input Manager..... 20
 Game Manager..... 22
 Physics..... 23

Room Class..... 24

Character Class..... 25

Object Class..... 26

Object Derived Classes:..... 26

User Interface..... 27
 Sphere of Interest..... 27
 Menu Manager..... 27

Menu Button..... 27

Menu Slider..... 27
 Inventory System..... 30

Scripts..... 30
 Console..... 30

Commands:..... 30
AI..... 31
Game Pseudo Code.....32
Main File..... 32
Game Loop..... 32
Tools..... 35
File Formats..... 38
Installer/Uninstaller..... 39
Team Sign-off..... 40
Peter Dunshee - Producer/Programmer.....40
..... 40
..... 40
..... 40
..... 40

Introduction

Hi-Concept

A 3D puzzle adventure game set in a mysterious mansion with quirky characters and a humorous storyline.

Overview

Operation: Stop Core is a humorous 3D puzzle adventure game that puts gamers into a whimsical world where collecting items and solving puzzles are key to winning. Selecting from a group of five characters with varying abilities, the player chooses two in which to explore a lonely mansion out in the countryside. While exploring this mansion, numerous items are collected to solve amusingly odd puzzles, and while doing so, a nefarious plot will be uncovered. Targeted toward the 13 - 35 year old demographic, Operation: Stop Core is to be a multi-platform release on Windows, Linux and Macintosh with an ESRB rating of E.

On starting the game, a group of friends are found driving out in the countryside. A billboard passes by advertising "Evil Chef's Mansion, Next 5 Exits". Realizing that they are all hungry and since Evil Chef's Mansion sounds so inviting, they decide to pull off of the highway. Two of the characters are sent off to get some grub while the rest hang back at the car. However, at the mansion things soon take a turn for the worse when one of the two friends falls down a trap door and becomes trapped inside the mansion. Now the one character must be rescued and, in the process, an evil plot is uncovered.

Operation: Stop Core pulls elements from several other puzzle adventure games to create a better gaming experience. The camera setup is similar to that used in Luigi's Mansion, in which all rooms in the mansion are viewed from the side. It is in 3rd person view with the camera following the player around the screen. The gameplay, however, resembles games such as Maniac Mansion, Grim Fandango and Escape From Monkey Island, where the focus is on exploration and puzzle solving. At the start of the game, the player chooses two characters with which to play through the mansion. For the player to solve puzzles and ultimately beat the game, the player must utilize both characters' abilities. A context-sensitive approach is to be used to allow the player to easily interact with the game world. In addition to gameplay, humor is a vital component in Operation: Stop Core. Much of the enjoyment of Operation: Stop Core revolves around humorously absurd characters, odd puzzles and a whimsical story.

All of these elements are intended to bring about a unique gaming experience for the player. Item collection and puzzle solving will provide a challenge that players will find engaging. The humor is intended to reach a wide audience, and so must be acceptable for all ages. This also means that Operation: Stop Core has a mix of humor, jokes and such that kids would enjoy as well as jokes that adults would understand. All of these, plus a fully interactive world, will act together to immerse the gamer.

Technical Goals

We have several goals for Operation: Stop Core. First is to make a fully 3D game with interactive environments. We also are making the game cross-platform.

Minimum System Requirements

1GHz Pentium III
128 MB RAM
Radeon 7500 or better 32 MB Video Card
75 MB Hard Drive Space

Recommended System Requirements

2 GHz Pentium IV
512 MB RAM
GeForce 4 32 MB or better
100 MB Hard Drive Space

Operation: Stop Core will run on the following operating systems:
Windows 9x/NT based OS, Linux and Mac too.

Risks

Fully 3D game: This is our first 3D game so there will be some learning involved in the process of getting things set up.

Cross-Platform: Making OSC cross-platform adds some complexity to the project.

Outside Tools

Development Environment

We are using Borland C++ Builder X. Borland's development environment works on Windows ®, Linux ® and Solaris ®.
Development in Mac OS X will be done in Apple's Xcode development environment.

Version Control

CVS will be used for version control. The repository is on the Producer's personal server.

Other

L-Forum: A web forum on the C-C-C-C-Combo Breaker website used for increased communication.

Timesheet: A PHP application used to help keep track of the hours worked by the team. It also keeps track of what the team member has been working on as well as a comment box for specific comments.

PHPBugTracker: Allows known bugs to be posted so everyone can look at them and find out who should fix them.

Development Team

Producer/Programmer – Pete Dunshee

Main programming tasks:

- Python game structure
- Python AI scripting
- Sound Manager
- Console

Technical Director/Programmer – Brian Eberspacher

Main programming tasks:

- Menu system
- Physics
- Window set up
- Graphics core

Designer/Programmer – Peter Young

Main programming tasks:

- Model loading/animating
- Room/World editors
- Room loading
- Special effects

Product Manager/Programmer – Robert Hunt

Main programming tasks:

- Input Manager
- Inventory system
- Puzzle scripting

Milestones

Engine Proof - 11/15/04

- Core Graphics setup
- Models loaded and animated
- Rooms can be loaded and displayed
- Inventory system main functionality
- Initial dialogue and text output
- Character collides with world
- Most menu functionality (minus graphical prettiness)
- Sound effects and music working
- Input complete
- Script Console

First Playable – 12/6/04

- Character walking around in a room
- Multiple rooms and transitioning from one to another
- Player can switch between characters
- Characters can identify objects in a room
- Characters can pick up InventoryObjects and have them be displayed in their inventory
- Character can interact with an object and use a command on it
- Player will be able to complete at least one puzzle
- Saving/Loading
- Actual dialogue

Pre-Alpha – 2/14/05

- Mansion built
- Most puzzles implemented
- Most cinematics

Alpha – 3/8/05

- All puzzles complete
- All cinematics
- Core gameplay done
- Miscellaneous physics objects complete

Beta – 4/4/05

- Bug testing
- Revise gameplay and 'fun factor'
- Shadow mapping

Gold – 4/18/05

- Marketing materials and packaging
- Bug fixing

Coding Guidelines

General Project Guidelines

- No #including in .h files
- Avoid defining functions in .h files
- No globals
- Use the typedefs providing for standard types (u32, s32, u8, s8 etc.)
- Do not use NULL, initialize pointers to 0.
- Use explicit casting, ie. static_cast, reinterpret_cast, etc.
- Rather than returning 0, -1, false, or any other random error behavior, throw instances of exception classes
- To ensure full cross-compatibility, code must ultimately compile with no warnings of any kind, on the highest warning level possible.
- At the top of every file you create, include a comment header with your name, date, and a brief description of the file's purpose
- At the top of every function you define, include a comment header with your name, date, and a brief description of what the function does
- If you change the purpose of a function you define, change the function header!
- ANSI style indenting will be used, four space indents
- Curly braces on their own line, with no indent
- Comment! Rule of thumb: if you have to justify an explanation for an algorithm to yourself, write it out in a comment

Variables

- Local Variables
 - Start with a capital letter
 - Capitalize every word in variable name
- Loop Variables
 - One-character names recommended
 - Start with 'i', then 'j' etc.
 - Use 'x', 'y' and 'z' with loops dealing with positions
 - If operation is not obvious, use descriptive name; Ex BitmapIndex
- Pointers
 - Prefixed with a 'p'
 - Ex: pLevel, pCamera, pPlayer
- Functions
 - Same as local variables
- Function Parameters
 - Prefix with a 'p_', then a variable name
- Classes
 - Prefixed with a 'c' Ex: cOpenGL, cVector3

- Member variable are prefixed with a `m_`

Game Mechanics

Game Core

The core of the game will be written in Python. At the start of the game, the game core will initialize every Manager. The Managers include Graphics Manager, Sound Manager, Input Manager and Game Manager. After each Manager is initialized the game core will run the primary script that will run the game.

Graphics Manager

The Graphics Manager will be in charge of setting up and keeping track of the game window and OpenGL. Both of these will be run through SDL. The Graphics Manager will load all models and textures in so they can be kept track of. We will be using OpenGL v1.2 and SDL v1.2.5.

Member Variables

- MainSurface – SDL main window surface
- CurrentDisplayMode – The current width/height/bit depth of the game
- DisplayModeList – A list of supported displaymodes
- Camera – The current camera
- TextureManager – A list of all the allocated textures
- ModelManager – A list of all the allocated models

Initialization

On start up, the Graphics Manager will create a full screen window, initialize OpenGL and set up OpenGL render variables. If either the window initialization or OpenGL initialization fail, it will cause the game to quit and display an error message. The Graphics Manager will start out with menu render variables. It will also enumerate the display modes available for the menu.

Start Game

When the game starts, the Graphics Manager will set up OpenGL variables for in game effects. It will set the right lighting variables, set up the stencil buffer for shadows, the correct alpha blending and depth testing setting as well.

Game Loop

During the game loop, the Graphics Manager will set up the initial OpenGL variables and then call the draw function of the current room, which will in turn draw the room and everything inside of it.

Termination

On termination, the Graphics Manager will release all textures, models and be in charge of destroying the window.

Object Models, Character Models and Animation

File Formats

For all models in the game, a modified version of the Wavefront OBJ file type will be used. File specifications for the standard Wavefront OBJ file format can be found elsewhere, for instance, at www.wotsit.org. The additional information to be stored in this modified OBJ file is a list of key frames of animation for the model as well as bounding box information. All models within the game will use this file type; non-animating objects will simply have one frame of animation. Models created by a 3D modeling program will be exported as the standard OBJ files. Then all the OBJ files that compose the animation of a model will be merged together to form one modified OBJ file, a .MDL type. A tool needs to be written in order to merge these files into the specification provided further down. The main reason for using this modified version is that it is in ASCII format, which eliminates many of the obstacles raised by this being a cross-platform endeavor.

The following is the expected format for the ASCII model file (expected tags are in < > brackets):

[modelName].mdl

```

<MODEL>           //indicates that this file contains model information
<VERSION> #       //gives the version number of this file (should be 2)
<NUMFRAMES> N     //this model has N frames of animation
<FRAME>          //Indicates the start of new frame data
.
.                 //one frame that follows the standard Wavefront OBJ format specification
.
</FRAME>         //indicates the ending of frame data
.
.                 //possibly multiple frames
.
<BBOX>           //indicates the start of new bounding box information (there can be
                  multiple bounding boxes)
(Cx,Cy,Cz)       //corner of bounding box
(Lx,Ly,Lz)       //length vector of bounding box
(Wx,Wy,Wz)       //width vector of bounding box
(Hx,Hy,Hz)       //height vector of bounding box
.
.
.

```

Each frame will basically just be a standard Wavefront OBJ. For purposes of animation, each frame is expected to contain the same number of vertices. Following the frames will be the bounding boxes. These are specified by giving a corner point and three vectors: a length vector, width vector, and height vector. This will allow for rotated or skewed bounding boxes, i.e. bounding boxes that are not axis aligned.

Internal Format

The Model Struct and Supporting Structs

```
struct RGBColor
{
    float r, g, b;
};
```

```
struct Point3D
{
    float x, y, z;
};
```

Defines a 3 value vertex.

```
struct TexCoord
{
    float u, v;
};
```

Defines a texture coordinate. This may or may not be necessary for a given object since some objects may not have textures.

```
struct TriFace
{
    ushort vertIndex[3];           //3 indices into a vertex array defining this triangle
    ushort tcIndex[3];            //3 indices into a tex coord array, tells which texture
                                   //coordinate to use with which vertex

    RGBColor color;               //color of the triangle
};
```

Defines a triangle face.

//Defines the animation states a model may have

```
enum AnimState
{
    RUNNING_STATE,
    USING_STATE,
    TALKING_STATE,
    IDLE_STATE,
};
```

```
class Model
{
private:
    UInt32 m_NumFrames;
    UInt32 m_NumVertices;
    UInt32 m_NumTexCoords;
    UInt32 m_NumTriFaces;
    vector<Vertex> m_VertArray; //vertex array
    vector<TexCoord> m_TCArray; //tex coord array
    vector<TriFace> m_TriArray; //triangles array
};
```

```

public:
    //basic constructor
    Model();

    //constructs model from given file
    Model(const char* filename);

    //loads the given file into the model
    void Load(const char* filename);

    //draws the given frame number without texturing
    void Draw(UINT32 framenum);

    //draws the given frame number with the given texture
    void Draw(GLuint texID, UINT32 framenum);

    //draws the given amount into the given animation state
    //with texturing
    void Animate(GLuint texID, AnimState state, UINT32 rangeVal);

    //draws the given amount into the given animation state
    //without texturing
    void Animate(AnimState state, UINT32 rangeVal);
};

```

All 3D models in the game will use this class. Static models, if any, will simply just have one frame of animation.

The Model Manager

To keep track of all the models used in the game, a Model Manager is to be implemented. Each model that is loaded up will be assigned a key so that if a model is used multiple times, only one copy will actually ever reside in memory. When a request is made of the Model Manager for a model, it checks whether that particular model has already been loaded. If not, it will load the model. Then, in either case, it should return the key for the requested model. The filename of the model to load will also act as the key for the model.

```

ModelManager
{
    public:
        //functions to add or remove models (filename is the key of the model)
        void AddImage(const char *filename);
        void RemoveImage(const char *filename);

        //Draws the Model with the given filename/key
        void DrawModel(const char *filename);
};

```

Note: Most of the Model and ModelManager functionality is already written.

Textures and The Texture Manager

Texture Class

This is a simple class that contains a single texture. Basically, it will just be a wrapper around an OpenGL texture ID. The Texture class will be handling loading up of image files and converting them to OpenGL textures. Textures are expected to be in 32-bit PNG (Portable Network Graphics) format, dimensions should be square, and dimensions should be powers of 2, not to exceed 256x256.

```
class Texture
{
    private:
        GLuint textureID;

    public:
        Texture(imagefile);    //construct texture from given image file
        GetTextureID();       //returns the texture ID
};
```

Texture Manager Class

To keep track of all the textures used in the game, a Texture Manager is to be implemented. Each texture that is loaded up will be assigned a key so that if a texture is used multiple times, only one copy will actually ever reside in memory. When a request is made of the Texture Manager for a texture, it checks whether that particular texture has already been loaded. If not, it will load the texture. Then, in either case, it should return the key for the texture requested. The filename of the image to load will also act as the key for the texture. To speed up rendering, the manager will allow access to a texture's OpenGL texture ID. This is so that a call to the manager is not necessary each time a texture is needed to be rendered.

```
TextureManager
{
    public:
        // functions to add or remove textures (filename is the key of //
        // the texture)
        void AddImage(const char *filename);
        void RemoveImage(const char *filename);

        //gets the OpenGL texture ID of the image with the given //
        // filename/key
        GLuint GetGLTextureID(const char *filename);
};
```

Note: Most of the Texture and TextureManager classes have already been written.

Camera Class

Used for the gluLookAt function to define what can be seen. The camera will have functionality for moving, and rotating in various ways. The only thing the camera needs to do is follow the player at a designated distance away. A simple translation method will be needed.

Member Variables

- Position – The position of the camera
- LookAt – What point is the center of the view
- UpVector – What direction is ‘up’, unit vector

Text

The text class is already written. Displays text on screen, nothing fancy.

Lighting

One goal for Operation: Stop Core is to have realistic lighting in the world. The only light in the world will come from actual lights in the room. If a window is in the room, a very dim ambient light will represent the moonlight.

Since realistic lighting is one of our unknowns, it will be classified as a lower-level feature. As we learn more about how the lighting will work, the appropriate text will be added here.

Sound Manager

The Sound Manager is currently done, but will be explained anyway.

Sound was written using `SDL_mixer`, which incredibly simplifies everything involved in playing sounds. Mixer uses standard SDL sound to accomplish its work; this means that all sound is automatically multithreaded, and playback doesn't need to rely on the timing system in the main loop. Trigger a sound and it will automatically mix and play back through an available sound channel of its choosing.

Here are the functions inside the Sound Manager:

`void Init();` - Initializes the sound system (completely, including sdl sound)

`void Update();` - Performs minor maintenance that is not handled in the sound thread, such as music fading when it is necessary

`void Quit();` - Finalizes the sound system and frees all memory

`void Reset();` - Finalizes and reinitializes the sound system

`bool CacheSound(const string filename);` - Pre-caches a sound effect into the sound system

`bool PlaySound(const string filename, int volume = MIX_MAX_VOLUME);` - Plays a sound; if the sound is cached it will play instantly with no latency. If it was not cached previously, the sound is cached by this function before playback

`bool PlaySoundPanned(const string filename, Uint8 leftpan, Uint8 distance);` - Same as `PlaySound`, except that you can specify a pan position and distance from the ear.

`bool PlaySoundPositional(const string filename, Sint16 angle, Uint8 distance);` - Same as `PlaySound`, except that you can specify an angled position and a distance from the ear.

`void PlayMusic(const string filename);` - Starts a music file; if one is already playing, begins fade out to fade in process

`void SetSoundVolume(int volume);` - Sets the master sound volume

`void SetMusicVolume(int volume);` - Sets the music volume

`void StopMusic();` - Stops the music from playing

`void KillMusic();` - Stops the music and frees the data it occupies

`void KillAllSFX();` - Stops all sound effects and frees all the cached data

Initialization

The Sound Manager will call its `Init` function which sets up SDL sound, sets the mixing frequency, sample format, number of channels the mixing chunksize, and allocates a reasonable number of channels

Start Game

When the game starts the Sound Manager loads the music for the two characters and the sound effects that are needed.

Game Loop

During

Termination

When the program ends the Sound Manager stops all sound effects that are playing, the music that is playing, and quits the SDL sub system.

Input Manager

The input manager “watches” the state of all input devices currently connected to the computer. Supported devices include: keyboard, mice, game pads, and joysticks. Each frame, the input manager queries each device, updates its own buffers, and readies itself to accept key/button/axis queries via functions callable by any other module. The input manager also includes a Python extension to allow Python scripts access to the input state.

- Keyboard – Fills an array with the up/down state of each key every frame. Uses this information to determine triggered states as well. The functions “IsKeyDown” and “IsKeyTriggered” allow access to this information.
- Mouse – Queries SDL Input for mouse button events. And stores them each frame.
- Game pad/Joystick – Queries SDL Input for button events, and stores them for access via the “IsButtonDown” function. Also queries for axis movement, which is stored in a floating point value, indicating how far from center the axis is.

Key mappings

The five control buttons are mapped into the following struct:

```
struct PlayerKeys
{
    u32 Action;
    u32 Cancel;
    u32 Inventory;
    u32 Switch;
    u32 Menu;
};
```

Each of the five unsigned 32 bit value can be either a keyboard key, a mouse button, or joystick/game pad button. Keyboard keys are mapped to the values 0 – 399, mouse buttons are mapped to the values 400 – 419, and joystick/game pad buttons are mapped to values above 420.

The states of the mapped keys are available through the “IsKeyDown”, and “IsKeyTriggered” functions. (Usage detailed below.)

Methods

bool IsKeyDown(u32 p_Key);

Returns true if a key is down, false otherwise. Possible values for p_Key are values from the SDLK enumeration. (SDLK_SPACE, SDLK_RETURN, etc.) Also the user may pass in a #defined keymap value to query for the player mapped buttons. (these values are #defined higher than any key on the keyboard, so no conflicts will occur.) Examples include: SC_BUTTON_ACTION, SC_BUTTON_MENU.

bool IsKeyTriggered(u32 p_Key);

Returns true if a key is triggered, false otherwise. Triggered state is determined by comparing the current down state of the key with the previous frame's state. (Triggered = IsKeyDown(This Frame) && !IsKeyDown(LastFrame)). See IsKeyDown for possible parameter values.

void Bind(u32 p_Key, u32 p_Button);

Binds the specified key to the specified mapped button. Checks the values of each to ensure their validity, then sets the value in the PlayerKeys struct to the new value. (Example bind(SDLK_SPACE, SC_BUTTON_ACTION).)

Game Manager

The Game Manager is in charge of keeping track of everything in the game world. This includes the players, the AI, all the objects in the world and each room, whether or not the game is in a menu or in game, and all of the physics calculations.

Member Variables

InMenu – A bool to determine if the game is in the menu system.

Room list – A list of rooms currently loaded.

CurrentRoom – Pointer to the room currently being displayed in the game.

Characters – A list of all the characters.

Initialization

When initialized, the Game Manager will set the game in the menu state and load the menus.

Start Game

The Game Manager will set up the game state depending on whether a new game is started, or an old one loaded. It will set up the two playable characters, the room to start in and set the AI in motion.

Game Loop

During the game loop the Game Manager will be in charge of running the player's update functions. If the player is in 'moving' mode, it will check the input and move the player. After the player moves, it will check for collisions and update the player accordingly. Then it will take the current player and see what objects in the room the player can interact with. It will also check to see if the player wants to bring up the menu and if so, sets up the options menu.

Termination

When the program ends, the Game Manager will call the cleanup methods of the rooms, all the characters, and all the objects currently loaded in.

Physics

The physics in OSC will mainly consist of collision detection and resolution. The characters and objects will be considered boxes as far as collisions are concerned. Each bounding box will consist of a corner and 3 vectors, representing the width, height and length of the box. The bounding boxes will be OOBB's (Object Oriented Bounding Boxes). Each object will have a bounding box, as will each character. The collision of the room geometry is still undecided. More than likely since the room geometry is going to have a low number of polygons, it will be tested on a per-polygon level. This actually won't be so bad because the only objects that need to be tested against it are the characters. The number of polygons per room is projected to be less than 50.

Box-Box Collision

To check if two boxes are colliding, the Separating Axes theorem will be used. At most, if the boxes are colliding, will require 15 checks.

Player-Object Collision Resolution

If a player collides with an object, all that will happen is to not allow interpenetration of the object and the player. Using the separating axes theorem, it can be determined where to place the player to just outside the other object. Since the player will be placed against the plane of the object it collides with, this will cover the floor and stairs as well. There might be some cases however, where there is no railing between the player and a drop. If a player 'drops' the physics will check how far the drop is, if it is less than a certain amount (to be determined how large this amount is), then the player will be allowed to move and drop, otherwise, the player will not move and not drop as if there is an invisible wall there.

Rope Object

The rope object will use a spring-mass system to simulate a realistic rope. It will be able to swing, get blown around by wind and can allow for any number of fixed points which aren't affected by gravity. If you were to hold a rope, the section that you are holding would be considered a fixed point. The rope object is partially complete. This is experimental and not critical for finishing the game.

Cloth Object

The cloth will be similar to the rope except that instead of a 1D array of spring-masses, it will use a 2D array. The cloth can be blown around in a wind and be held up and any point or number of points, like a curtain. This is also partially done. This is experimental and not critical for finishing the game.

Room Class

The room class will contain everything necessary for a room. It will contain the room geometry, a list of all Objects in the room, a list of all characters in the room and a list of lights in the room.

Member Variables

- RoomModel – A model that will draw the room.
- ObjectsList – A doubly-linked list of all the objects. We're using a list because objects can be added and removed during the game, so this is the most efficient container. These are pointers to objects and not objects themselves.
- CharacterList – A doubly-linked list of all the characters in the room. These are pointers to characters.
- AmbientLight – The ambient light of the room.
- LightList – A list of all light objects, these contain pointers to all the light objects in the room. Same light objects that are in the objects list.

Update

When a room is updated, its first task is to insure that all objects and characters are in fact, still inside of it. It does this by comparing their room pointers against itself. This is just in case a character or object leaves a room and doesn't tell the room that it left. Then it calls the update functions of all characters/objects inside of it.

Draw

During the room's draw function, it will first draw the room geometry, then call the draw methods of all objects inside of it, then call the draw methods of all characters inside of it.

Character Class

The character class will hold all the information needed for any of the main characters, playable or not. The character class contains a pointer to a mesh to draw the character, as well as the character's orientation, position and all the animation info the character needs. The character has an inventory of all objects it is holding. Additionally, it will hold a pointer to the room this character is in.

Member Variables

- UpdateState – This is the state of the character used to determine how to update it. Different states include:
 - CHARACTER_STATE_MOVE – Used when the character is moving around a room and needs to either check for input, or let AI control it.
 - CHARACTER_STATE_EXAMINE – Used when a character is examining its Sphere of Interest. This is so the Game Manager can update differently.
 - CHARACTER_STATE_INVENTORY – Used when in the inventory.
 - CHARACTER_STATE_TRADE – Used when a character is trading items with either another character, or an object (like a drawer)
- MovementState
 - MOVEMENT_STATE_IDLE – Uses the idle animation.
 - MOVEMENT_STATE_MOVING – If a character is moving, its walking/running animation should be used.
 - MOVEMENT_STATE_USING – Uses using animation.
 - MOVEMENT_STATE_TALKING – Uses talking animation
- Inventory – A list of pointers to all of the objects the character has.
- Orientation/Position – What direction the character is facing and where they are, the position is in world coordinates.
- Animation – What the current animation of the character is.
- BoundingBox – The bounding box of the character.
- Model – Pointer to the model that represents this character.

Update

The update function for a character updates the character's animation.

LookForObjects

When a player looks for objects, it will take its position and use a sphere to find the objects that are close to it. This is really easy, just check to see if the distance squared of the player/object is less than the radius squared of the sphere.

AddObjectToInventory

This takes in an InventoryObject and adds it to the player's inventory. This method should also remove that object from the object list that it was obtained from.

Character State Machine

The character state machine should be written in way to allow key input to change the player's state and to allow the AI to control a character easily as well. Each character will use movement states to control how they can move. Our character's state machine is relatively simple; the only true movement states are idle and moving. The only way to enter the use or talking states is through using objects or items, or through a scripted scene. There will be set methods to set the character's animation for each state, which will be sufficient to handling the character's states.

Object Class

Objects are things in the world that can't be picked up, but can be interacted with. Examples would include windows, a telephone, a radio or anything that an action can be used with. Each GameObject will know what actions can be used on it. Since there are many different GameObjects, this will be a base class. If there is a specific GameObject with some random behavior, then it can be in the form of a derived class.

Member variables

- CanUse – Whether or not the Use command can be used
- CanTake – Whether or not the Take command can be used
- CanOpenClose – Whether or not the Open/Close command can be used
- CanTalk – Whether or not the Talk command can be used
- UseItemWithList – A list of items that can be used with this Object.

Object Derived Classes:

- cObjectContainer – A cObjectContainer is an Object that holds other Objects. These all have Open/Close methods that allow the player to trade inventory between the cObjectContainer and the character.
- cObjectLight – A cObject that emits light. The Use command causes the light to be turned on/off.
- cObjectActivator – An activator is an object that when Used, will activate a command on another object. It will contain pointer to the object(s) that it activates and a define for what command it uses on the object(s). This will be used for levers and switches.
- cObjectBookShelf – A special object that when 'used' will cause the book shelf to be knocked down. The bookshelf will have an animation that does this.
- cObjectDoor – A door when opened or used, will transition the player to another room
- cObjectGuestBook – When used or opened, will bring up the save screen.
- cObjectGuillotine – More complicated because when used with an item, the item is placed on the guillotine, or if used near the base, the character will lie down on it. If used near the blade, will cause the blade to drop. The guillotine will need to know if there is an object or character on it so that if the blade is dropped, can cut whatever is on there.
- cObjectIronMaiden – The Iron Maiden if used while the door is opened, will make the character get inside of it. If open and used with an item, will put the item inside. Then if an object or character is inside, and the door is closed, will close the door on the item/character. This needs an animation for opening/closing.
- cObjectKnifeRack – If used, activates the knife rack puzzle
- cObjectOldCar – If used, activates the car puzzle
- cObjectDanceFloor – If used, activates the dance floor puzzle

- cObjectStove – Using turns the pilot light on/off. Using item with while the pilot light is off will try and burn the item.
- cObjectStairs – Stairs are a fairly special type of object, they need to allow the player to walk up them. The room editor will allow the placement of a collision box along the plane that the stairs go up, so collision will be fairly cheap.

User Interface

Sphere of Interest

The Sphere of Interest (SOI), is used to tell what the player can interact with. If an item is inside the SOI, pressing the 'Examine' key will allow the player to examine the item and then use a command on it.

Determining if an Item is within the SOI

The SOI will have a radius, to determine if an item is within the SOI, the distance squared from the item to the sphere's center should be less than the radius squared.

Item behavior inside the SOI

If an item is inside the SOI and is being examined, the currently focused item will be flashing. When an item is flashing, it will start to be blended with white with the white having an alpha of 0, to being interpolated until the white has an alpha of 255, to back down to white having an alpha of 0.

Menu Manager

The menu manager will manage the menu system. It will hold all of the menu screens and will keep track of the current menu screen being displayed and how it is traversed. It will utilize a stack to keep track of going down into the menus so it can correctly back out to the main menu. This is used because one menu screen can be accessed from several different screens, so hard coding where the 'back' function takes you, is not an option. The Menu Manager will actually be contained inside of the Game Manager to help separate being in a menu and being in game. The input for the menus will be described as follows, arrows move the selection, the 'action' key selects and action and the 'cancel' key goes back, or unselects an option.

Menu Button

A button on the menu used to select an option. It will display text of the name of the option and if selected, will activate it. Each menu screen will have a list of buttons of all the choices available.

Menu Slider

The menu slider will be used when there are many different values for an option. A good example of this is for music volume. When the slider is selected, the bar can be moved left or right along the slider axis. Pressing action while the slider is highlighted will select

it, then pressing left or right will move the bar. After selected, pressing action will change the value of the slider, while cancel will not change the value.

DigiPen Splash Screen

Description – Displays the DigiPen logo and its copyright information

Key Input – Action key moves screen to the Team Splash Screen

Team Splash Screen

Description – Displays the C-C-C-C-Combo Breaker logo and copyright

Key Input – Action key moves screen to the Main Screen

Main Screen

Description – Is the main screen, allows several selections.

Key Input – Arrows move in between selections. Action key selects options.

Selections

- New Game – Starts a new game, more info below
- Load Game – Sets current screen to Load Game Screen
- Options – Sets the current screen to Options Screen
- Credits – Sets the current screen to Credits Screen
- Exit Game – Exits the game

New Game

Description – Starts the New Game sequence, when that is done the Character Select Screen comes up.

Character Select Screen

Description – Shows all five characters in full 3D in their idle animations. When a character is highlighted, a description of the character will displayed. Pressing action will select the character, which will play their ‘happy’ sound effect. If one character is selected they will be moved forward so and won’t be able to be selected again. Pressing cancel will unselect the first character while action will select the second character. After two characters are selected, a confirmation will appear with ‘yes’ or ‘no’ options. The user can highlight either and press action or can press cancel and go back. Not continuing here will unselect the second character only. If the user confirms their choice, the game will start.

Key Input – Arrows switch between different characters, action selects a character, cancel unselects a character.

Load Game Screen

Description – Shows the guest book with all the saved games on it. Each saved game will show the two characters selected as well as time played.

Other – If a game is selected, a confirmation box will appear. ‘Yes’ loads the game, ‘No’ goes back to the Load Game Screen.

Options Screen

Description – Displays all the options for the game

Selections

- Video - Goes to the Video Options Screen
- Sound – Goes to the Sound Options Screen
- Controls – Goes to the Control Options Screen

Video Options Screen

Description – Gives all the options for changing graphical variables.

Selections

- Resolution – A slider to change the resolution.
- Shadows – Toggles shadows being on

Sound Options Screen

Description – Displays all the options for sound

Selections

- Music Volume – A slider to change the music volume
- SFX Volume – A slider to change the sound effects volume

Control Options Screen

Description – Allows the user to change the controls for the game

Selections

- Action – Map the action key
- Cancel – Map the cancel key
- Inventory – Map the inventory key
- Switch – Map the switch key
- Options – Map the options key
- Move Up – Map moving up
- Move Down – Map moving down
- Move left – Map moving left
- Move right – Map moving right

Save Screen

Description – Allows the user to save their game. Can only be reached by using the GuestBook at the entrance of the mansion

Selection

- One slot for each save file

Inventory System

The inventory system mainly needs to allow the drawing of objects in a wheel. The wheel will be in 3D and will be along the xz-plane. (The xz plane is the plane going from left to right and from front to back) The selected item will be in front of the player, slightly enlarged. Rotating the wheel should allow the user to see the rotation instead of just instantly moving the next object to the front.

Scripts

Console

The console's main purpose is for debugging. Certain commands will be programmed into the console to allow variables and states to be dynamically changed in game. For example, in order to load a different room, the user could type in `changeroom "room name"` where changeroom is an identifiable command and "room name" is a string (no quotes) that will be the name of the room to load. This way, we don't have to hard-code shortcuts or re-compile to change to a different room etc. In order to utilize the console, different extensions will have to be written for each manager. This is necessary so that Python can communicate with the manager to alter it. The functionality of the console will be flushed out as the development process continues.

Commands:

(todo: list all of the commands here as we make them up)

Sound commands:

cachesound
playsound
playsoundpan
playsoundpos
playmusic
setfxvol
setmusicvol
stopmusic
killmusic
killallsfk

Input commands:

Graphics commands:

Game commands:

AI

General AI will be done using generator functions in Python. These functions are relatively new to Python and are essentially resumable functions. This allows you, in essence, to roll a state machine right into the code that operates on it. All local variables in the generator are carried over into the next iteration, and the function takes up from where it broke off its execution last time.

Rowsdower has a couple of things that he can do. His basic AI is simply a generator that conditionally goes through a few different phases. He'll have some variables that fluctuate and cause him to get hungry or bored or whatever. These will cause him to go into different sub-generators which cause him to go into the kitchen to eat or into the living room to watch television. Different activities should cause different things to happen to the variables so that he seems to do different stuff. At any rate, Rowdower will keep on running this program until the game ends or until he sees Tim Bucktoo.

If Rowdower sees Tim Bucktoo, he will go into his other AI state until he can't see Tim anymore. Rowdower's other state will simply be a simple pathfinding algorithm that moves Rowdower closer to Tim Bucktoo. It won't require A* or anything complicated, because if Tim gets out of line of sight, Rowdower will go back into his wandering state.

Rowdower in his wandering state will have a node-based path system that allows him to get from where he is to any other room that he needs to get to.

The squirrel has very basic AI. When he is unleashed, as soon as a player tries to "use" or "pick up" anything, etc, the squirrel will fly onto the screen and kill him. The squirrel's generator will simply check for that state, and the room that it is happening in, and "teleport" him there, so to speak. After both players are killed and he is combing the mansion for the other NPCs, he will use the node path system to find the other characters.

Game Pseudo Code

Main File

```
int main( int argc, char *argv[] )
{
    // init SDL

    // initialize game core

    // initialize all managers

    // start main loop

    // quit program
}
```

Game Loop

```
void MainUpdate( void )
{
    // call the update functions of all the managers
    // first is the Input Manager since it needs to get the // new
    input state
    InputManagerUpdate();
    // after the input is updated, update the game since
    // most stuff happens in the game manager

    GameManagerUpdate();

    // next update the sound
    SoundManagerUpdate();

    // finally draw whatever the new scene is
    GraphicsManagerUpdate();
};

void InputManagerUpdate( void )
{
    // Check the new mouse and keyboard state
    GetMouseAndKeyBoardState();
};
```

```
void GameManagerUpdate( void )
{
// Most updating goes on in here
// check if in the menu
if( InMenu )
{
    UpdateMenu();
}
else
{
    // first check the main player's state, then
    // update accordingly
    if( PlayerState == MOVE )
    {
        CheckInputAndMovePlayer();
        // update the room which will in turn update // the
        // objects and characters inside the room
        // only update here because this stuff is
        // paused when in the other states
        UpdateRoom();
    }
    else if( PlayerState == EXAMINE )
        CheckInputAndExamine();
    else if( PlayerState == INVENTORY )
        CheckInputAndLookInInventory();
    else if( PlayerState == TRADE )
        CheckInputAndTrade( Inventory1, Inventory2 );
}
};

void SoundManagerUpdate( void )
{
    // the sound manager's update function is already // written
    SoundManager.Update();
}

void GraphicsManagerUpdate( void )
{
    // clear OpenGL backbuffer and depth buffer
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // call the game manager's draw function
    GameManagerDrawFunction();

    // swap the buffers
    SDL_GL_SwapBuffers();
}

void GameManagerDrawFunction( void )
```

```
{
    // check if in the menu, if so, call the menu manager's // draw
    function
    if( InMenu )
        MenuManagerDrawFunction();
    else
    {
        // need to draw the room
        CurrentRoomDrawFunction();

        // check the main character's state and then draw
        // anything else that needs to be drawn
        if( InInventory )
            DrawInventory();
    }
}
```

```
void RoomDrawFunction( void )
{
    // set up camera's view
    gluProject( camera );

    // set up lighting

    // loop through and draw all objects

    // draw all characters
}
```

This is how the game will be maintained on the surface. Of course there will be much more in all the underlying functions, but trying to plan everything out now would prove a waste of time since it will change and future problems will arise.

Tools

World Editor

Functionality Overview:

- construct room geometry
- place rooms in 3D world
- place objects within each individual room
- link rooms with doorways
- loading/saving of world files

Room Construction Functionality:

- place walls and define their length
- define height of room
- set textures for walls, floor and ceiling

Room Editing Functionality:

- place objects within a room
- allow to snap-to other objects (useful, for instance, when placing an object on a table)
- specifying which objects are interactive

Description:

The World Editor is necessary for the construction of the mansion. It will allow us to construct rooms, place objects within the rooms, place rooms within the mansion, link those rooms together by way of doorways, and provide saving and loading of world files.

For constructing room geometry, it allows walls to be placed together to form rooms. The length of individual walls should be able to be defined. In constructing the room, it is also necessary to set textures for the walls, ceiling and floor. Each room should have a unique ID in order to distinguish it from every other room.

Editing a room will allow objects to be placed inside it. For placing objects, it will be useful to have the option of "snapping" objects to other objects. This is useful for placing objects within relation to other objects. As an example, trying to place a lamp on a table would prove to be a challenge without this functionality. With using the snap-to option, the bottom of the lamp can be placed exactly on the top of the table. Another useful function of room editing is the ability to set whether an object is interactive or not. Since the game will check what is in range of the player to interact with, it will be useful to know what objects cannot be interacted with, thus eliminating some unnecessary intersection calculations.

One of the main functions of the World Editor is the placing of rooms. It should be just like placing objects within a room, except in this case it will be placing rooms within the world. Another vital function of the world editor is the placement of doors.

The editor will allow two rooms to be linked together by placing a doorway between them. The doorway will be a special object with a unique ID that resides in both rooms that are linked together. Finally, the editor needs to provide the ability to save and load world files to and from disk.

World File Format

The World files created and used by the editor will adhere to the following specification:

[filename].wld

```

<WORLD> //indicates that this is a WORLD file
<VERSION> [Version #] //gives the version number of this file (should be 1)

    <ROOM> //start of new room data
    [ID Name] //unique ID number
    (Px,Py,Pz) //room's position (Px,Py,Pz) in world coordinates

        <LAYOUT> //start of room geometry information and texture information

            <WALL> //start of wall data
            (x,y,z) (x,y,z) (x,y,z) (x,y,z) //four corner points defining the wall (relative to room
position)
            (r,g,b) //color of wall
            [Texture Filename] //filename of texture for the wall
            .
            .
            .

            <FLOOR>
            [Texture Filename] //filename of the texture for the floor
            <CEILING>
            [Texture Filename] //ceiling texture (floor and ceiling geometry is
determined by geometry of walls)

        </LAYOUT> //end of room layout information

            <OBJECT> //start of object data
            [ID Name] //unique ID of an object (all IDs should be found in the object
list further down)
            [Type] //0 - not interactive, 1 - interactive, 2 - a door, 3 - stairway
            (Px,Py,Pz) //object's position (Px,Py,Pz) relative to the room
            [S] //scaling factor S of object
            (Rx,Ry,Rz) A //axis (Rx,Ry,Rz) to rotate around by an angle A

            <OBJECT> //start of other object data
            .
            .
            .

    </ROOM> //end of new room data

    <ROOM> //start of other room data
    .

```

```
.  
.
<OBJECTS>           //starts the listing of all objects available in the world
[ID Name] [Filename] //object's unique ID followed by the filename of the object
.
.
.
</OBJECTS>         //end of object list

<DOORS>            //starts list of doors
[Door ID] [Room ID] [Room ID] //unique ID for the door and the IDs of the two rooms it links
.                          //(doors are considered to be objects within a room)
.
.
</DOORS>
[End Of File]
```

Rooms will contain data describing the room geometry and objects found within the room. The room layout specifies the position of walls, textures and colors for walls, and textures for the ceiling and floor. Objects will have a position relative to the room, a scaling factor, an axis-angle pair for rotation, and a value specifying the type of object: interactive, non-interactive, door, or stairway. Objects will only be known to the room by an ID; the complete list of object IDs and their respective files is stored by the world. Doors are a special object that link two rooms together. Stairs are also a special object that allow the player to walk up and down them.

Note: Brackets, [], are not to be included in the file. Also, it is not necessary to indent anywhere in the file, white space will be ignored.

File Formats

Models – .mdl

Textures – All textures will be .png

World - .Our own .wld format

Music - .ogg

SoundFX – low quality .ogg, still deciding

Installer/Uninstaller

On windows, we will use the Inno installer, which is very easy to use and merely involves putting together a simple little script which compiles the installer exe file. On linux, we will use the Loki installer, which is also very easy to use. Along with the Loki installer, we will use the makeself script system which can compile all of the Loki installer files, etc, into a single self-extracting shell script. On Mac OS X, we will use a compressed disk image file. The EULA can be attached to the disk image so that when it is launched, the license is displayed and must be agreed to before it can be mounted.

Team Sign-off

By signing here, you agree to follow the TDD or something.

Peter Dunshee - Producer/Programmer

Brian Eberspacher – Technical Director/Programmer

Peter Young – Lead Designer/Programmer

Robert Hunt – Product Manager/Programmer
