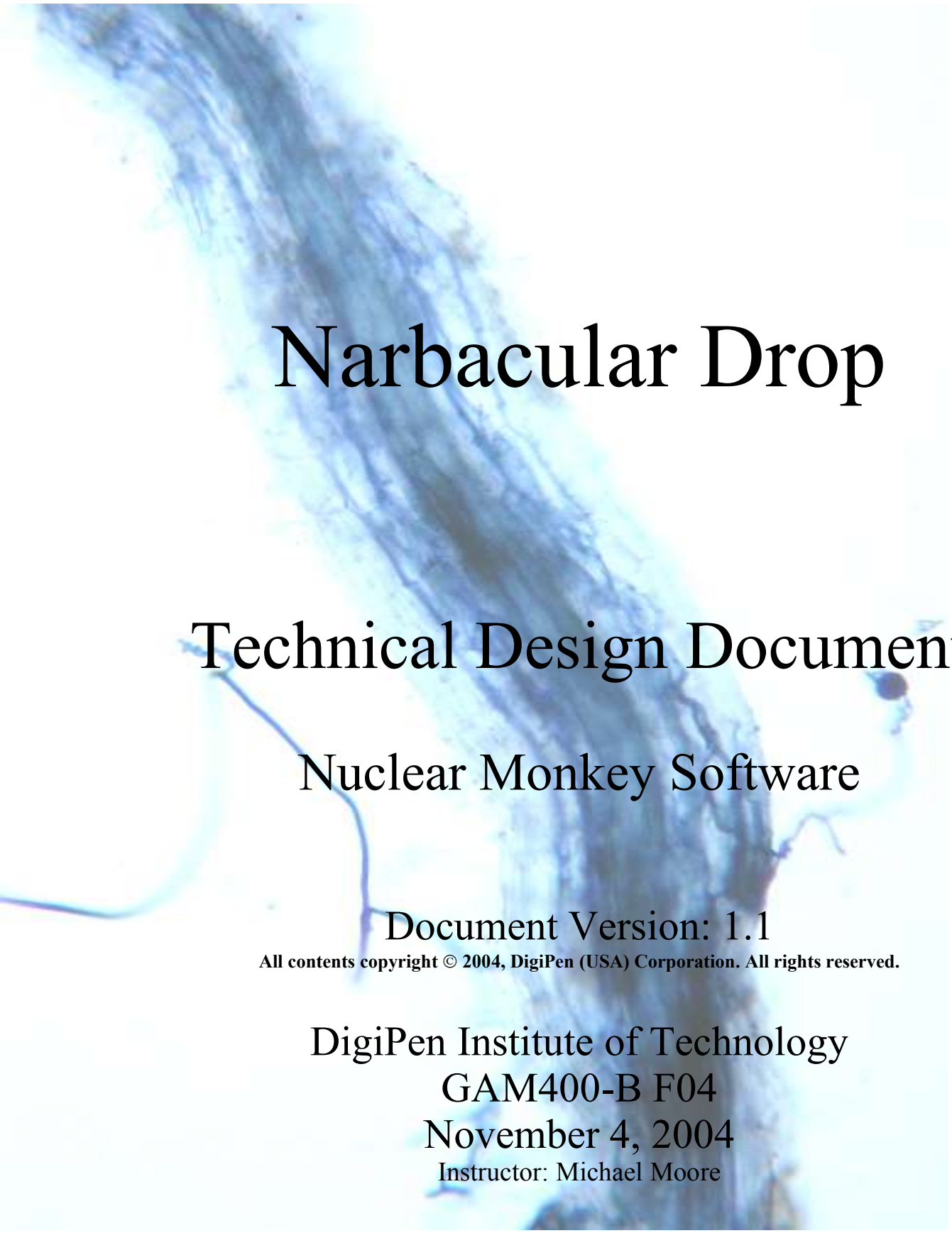


Narbacular Drop

Technical Design Document

Nuclear Monkey Software

Document Version: 1.1

All contents copyright © 2004, DigiPen (USA) Corporation. All rights reserved.

DigiPen Institute of Technology
GAM400-B F04
November 4, 2004
Instructor: Michael Moore

Table of Contents

Project Overview

Game Concept

The player is a princess stuck in a dungeon-like atmosphere where she must solve puzzles with the use of her ability to summon a single set of linked portals on earthen walls and floors/ceilings.

Technical Goals

A-Level:

- Lightning fast 3D graphics. (Already ~80% complete)
- Seamless portal technology on axis-aligned planes.
- Immersive 3D ambient sound. (Engine portion already 100% done)
- User-Friendly interfaces.
- Realistic Newtonian physics. (Already ~100% complete)
- Animating Meshes. (Already 100% Complete)
- Component Meshes.
- Basic particle systems.
- Simplistic A.I.

B-Level:

- Normal based bump mapping. (Already 60% complete)
- Believable A.I.
- Rigid body physics with lava. (Already 20% complete)
- Optimized particle systems.
- User-Friendly 3rd person camera views. (Already 50% complete)
- Rotational inertia.

C-Level:

- Shadows.
- Expand portal technology to support arbitrary planes.

System Requirements

OS: Windows 2000 and XP

Minimum Hardware:

- 1050 MHz Processor
- 256MB RAM
- 3D Accelerator Card with Direct3D 9.0 support, 32MB onboard video ram, hardware transformation and lighting support.
- 300MB Hard-Drive space available.

Recommended Hardware:

- 1.8 GHz Processor
- 512MB RAM
- 3D Accelerator Card with Direct3D 9.0 support, 96MB onboard video ram, hardware T&L support, pixel shader 1.1 support, vertex shader 1.1 support.
- 300MB Hard-Drive space available.

Technical Risks

- Portal technology could break or be horribly slow on some hardware.
- Complications with portal technology could bring about large amounts of hacks, counter-hacks, and cross-hacks.
- FMOD could suddenly change their licensing and we'd have to make time to create a robust sound system.
- Lava might not cooperate.
- The entity system could have some crippling fault that we haven't thought of yet.

Third Party Tools

FMOD

FMOD is an audio library developed by Firelight Technologies for 2D and 3D sound as well as music. It comes with a free usage license for non-commercial projects.

http://www.fmod.org/

Worldcraft / Valve Hammer Editor

Worldcraft is a 3D level design tool owned and maintained by Valve Software. Its name has recently changed to "Valve Hammer Editor", but is still the same tool.

http://collective.valve-erc.com/index.php?go=hammer

Inno Setup

Inno Setup is an installer package developed by jrsoftware. Please refer to the “Installer” section for specific details on this program.

http://www.jrsoftware.org/isinfo.php

Game Play

Game structures / Game objects

The following is a list of logical objects currently planned for the game:

- Wall – A wall.
- Button – A button that you usually need to press to open a door.
- Static camera position + trigger – 3rd person fixed camera positions and the trigger areas for them.
- Door – The doors at the end of each level.
- Lava - …..Lava.
- Lava heater – The pits that heat lava.
- Crate – Generic crates that are nothing more than weights.
- Fireball – Demon's weapon of choice.
- Boulder – The primary obstacle for a few levels.
- Torch – Lighting coolness.

Characters:

- No-Knees: The main character that the player controls
- Demon: The end boss, big scary dude….fireballs…pain…
- Impy: The dungeon's 'janitor'. Intended to be a nuisance on some levels.
- Lava turtle: A ride-able turtle that swims around in lava.
- Wally: The entity of the dungeon, personified by a mouth that opens on the walls.

Physics

Our physics will be based on a simplistic Newtonian model.

Newtonian

A simple velocity and acceleration based physics model that handles gravity and simple collisions. This is used by objects to modify their position based on the internally managed velocity.

It contains:
- The mass (kg) of the object. This is used for collisions.
- The ratio of energy it loses on impact. So 0.1 is very rubbery and 0.9 is very sticky. Negative values will gain energy on impact.
- Current velocity (m/s). This is not normalized as it also has a magnitude.
- Speed that is automatically calculated as the magnitude of velocity.
- Gravity acceleration that is applied every frame. By default it's 9.8 (m/s^2).
- Accumulated acceleration is used for efficient processing of outside forces. Any calls to accelerate a Newtonian are summed in this variable and are used during normal updates once per frame.
- Number of collisions is the total number of collisions that have been reported and represents it as only one collision that frame.
- A ground Boolean that keeps track of if the Newtonian is resting on a surface. This activates a slightly different model that ignores gravity and uses friction.

An “Accelerate” function allows outside forces to act on the Newtonian. Newtonians can also be accelerated by using helper functions that handle physical collisions between each other and with a static plane.

When all accelerations are processed the velocity is capped at the maximum value 80m/s.

Physics Manager

This is a simple interface that manages a list of Newtonians. RequestNewtonian() Newtonian will add a new one to its list and return a pointer to it. Passing that pointer back into ReleaseNewtonian (Newtonian*) will remove that Newtonian from its list. Newtonians in the Physics Manager's list all have their Think()s processed.

Portal physics

Our portals effectively break 3D space and require special processing. This special processing takes effect by simply checking which side of a portal the object's center is on. Then applying physics based on that side being the "correct" side. Once an object's center passes the portal, then physics is applied as though the destination side were the "correct" side.

Projectiles

The only projectile in the game is Demon's fireballs. These will not be affected by physics and will simply travel along a straight line until they hit something.

Collision

Collision Manager

Handles collisions between Elimination Layers in a list. Each object is checked once against all other objects. This is handled so that there isn't an overlap such as object A checking against object B and then object B checking against object A. Collision Layers check against each other with a generic Test (EliminationLayer*) function that returns 1 for collision, 0 for no collision, and -1 for unknown. Objects that collide have a pointer to the opposing object added to their list of objects for that frame.

Collision Map Manager

Similar to Collision Manager, but checks the Elimination Layers in its list against a list of map geometry. This is an optimization that allows maps geometry Elimination Layers not to check against each other.

Elimination Layer

An abstract base class that contains a pointer the parent using the layer, a string table pointer that defines the type of layer, and a list of objects that have collided with the layer in the past frame. A pure virtual Test(EliminationLayer*) function is to be filled out by each deriving layer type. The test function should first determine the type of layer it is testing against. Then provide a test that checks between itself and that type, returning 1 for collision and 0 for non-collision. If the type is incompatible or unknown Test should return -1.

Player actions

The player will take on the body of princess No-Knees. And princess No-Knees is capable of the following actions:

- Creating 2 interlinked portals with the help of Wally.
- Walking around, including 3D strafing action!
- Looking around her environment.
- Falling
- Walking into a portal

It should be noted that the princess is NOT capable of jumping or pushing objects around the world.

Victory conditions

The player wins the game when Demon is dead. Each level is beaten based on its specific win conditions (elaborated in "Level Specific Code").

Level specific code

Level 1:
Specific code required:

Polishing Up

- Detection for when all 3 buttons have been pressed at the same time and the door needs to be opened.
- Specialized A.I. for Impy to control box polishing and retrieval.
- Wally hint conditions and sequences.

Level 2:
Specific code required:

Boulder Dash

- Boulder automation.
- Detection for when both buttons have been pressed at the same time and the door needs to be opened.
- Wally hint conditions and sequences.

Level 3:
Specific code required:

Hallway to Hell

- Detection for when the player is on the exit platform and the door needs to be opened.
- Wally hint conditions and sequences.

Level 4:
Specific code required:

Ledge Ladder

- Boulder generation/destruction and automation
- Wally hint conditions and sequences

Level 5:
Specific code required:

Fire with Fire

- Demon A.I.
- Lava pit raising conditions.
- Wally cheering sequences.

Artificial Intelligence

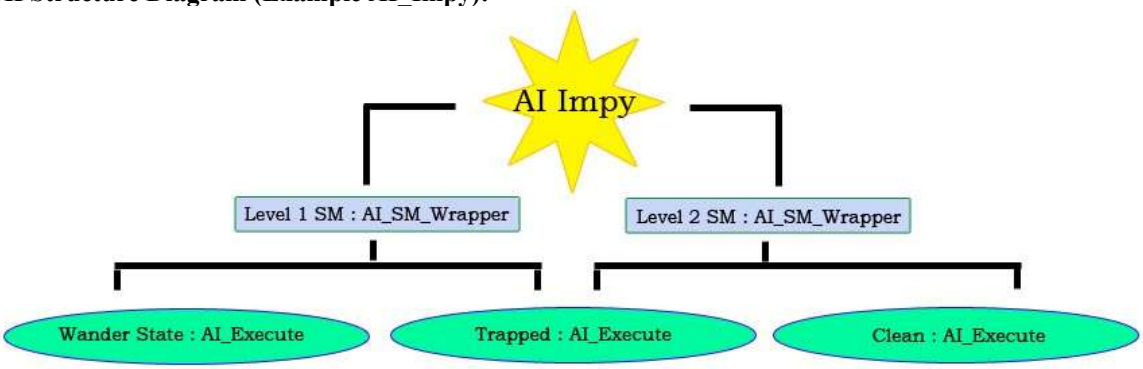
Artificial Intelligence for the game will be broken up into various state machines that can be attached to any game object. Game objects that will require AI are Impy, Demon, Wally, and Lava Turtle (these will later be described more in depth). Here is some various information that one would need to know about AI.

AI Structure:

Every object that is said to have AI, will have a set of C++ files (.h and .cpp) associated with it that will be the character's AI manager. For instance, Impy will have files called AI_Impy. These files should contain a class that has general information about the character, such as run speed, access to the character's model, and access to physics and collision objects. This class should also contain instances of the state machine class, and a way to manage which one should be in use, so that way you can have multiple state machines for one character.

- *State Machines at the base level:*
    - **Definitions**:
        - *Goals* – Goals are actions that will be added to a state machine's goal stack. These are objects that will be executed as a continuous action, and is a character's driving force at that time.
        - *Rules* – Rules can be either conditional, or an action. These are sub-states under every goal.
            - *Conditional* – This is basically a check to see if a particular action should be executed.
            - *Action* – This is a sub-state, which upon receiving the go-ahead from a conditional rule will be executed. These are really in fact references to already registered goals.
        - *Commands* – These are statements which will decide how a goal is executed if a conditional statement is met.
            - *GoTo* – This command will remove the current goal off of the goal stack, and replace it with the new goal.
            - *GoSub* – This command will keep the old goal on the stack and push the new goal on top of it.
            - *Return* – This command will pop the current goal off of the stack and revert to using a previous goal (this is the compliment to the GoSub command).
    - **Files**:
        - *AI_StateMachine* – This is base class that handles which rules and goals are going to be executed for a given state machine. AI_SM_Wrapper is derived from this class. It contains a
        - *AI_SM_Wrapper* -- This is a base class that wraps the AI_StateMachine and any character's state machine will be derived from this class. It contains a macro that should be filled out in the derived character state machine; this will create a new AI_StateMachine, and add goals and rules as necessary.
        - *AI_Goal* – This is a class that will represent a goal that a character will have, and that manages the addition of rules.
        - *AI_Rule* – This is a class that encapsulates a rule, and defines the various commands (listed above).
        - *AI_Execute* – This is a virtual base class that any character's states that will be executed should be derived from. It has functions to fill out for the initial action a state should have, a function that should be updated every time the AI is called, and a function for a state's final action.

AI Structure Diagram (Example AI_Impy):

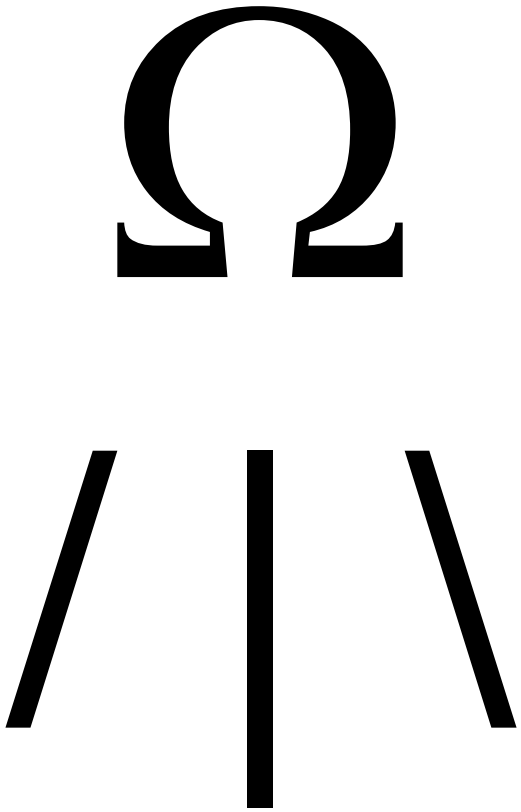


Game Object AI:

- *Impy*
  - **Structure**
    - Goal: Wander
    - Goal: Idle
    - Goal: Chase
    - Goal: Flee
    - Goal: Clean Box
    - Goal: Move Box
    - Goal: Block Portal
    - Goal: Tantrum
- *Demon*
  - **Structure**
    - Goal: Wander
    - Goal: Summon Boulder
    - Goal: Summon Fireball
    - Goal: Jump to Next Level
    - Goal: Gloat
- *Wally*
  - **Structure**
    -
- *Lava Turtle*
  - **Structure**
    - Goal: Wander
    - Goal: Flail
    - Goal: Idle

Multiplayer

Narbacular Drop will be a single player game only. There is no planned support for any type of networking or hot seat style swapping.



Code Overview

Main game loop

The main game loop exists within GDot's "Think" function. Every system besides the Sketcher Engine is a sub-entity of GDot. These systems will have their Think functions called in the same order that they were added to GDot. Each sub-entity of those systems will in-turn have their Think functions called.

The entity-think system is amazingly flexible in this way, allowing for whole systems to be disabled with a single line of code without crippling or crashing the game.

File formats

All code should be contained in standard C++ source files (.cpp) and standard C/C++ headers (.h) with DigiPen copyright information contained in comments at the top of the file.

The project workspace will be contained within a Microsoft Visual Studio .NET solution (.sln).

Levels will be contained within "compiled map format" files (.cmf), which are exported from Worldcraft's map files (.map). More details can be found in the Worldcraft appendix.

Please refer to the graphics and audio sections for their specific file format requirements.

Comments

Single line comments should be used as often as humanly possible to elaborate code flow.

Function prototypes should include a single line comment summarizing their purpose. Function bodies should provide a doxygen compatible comment header.

Specific marker tags:

- "TODO:" should be left in places where the coder is skipping small functionality for the time being to get a major feature done first. TODO's should elaborate what features need to be implemented later.
- "RECODE:" should be left in places where the coder knows they're being sloppy. Recodes should contain suggestions for better code if ideas are available, but at the very minimum should point out specifically what is sloppy.
- "HACKHACK:" should be used whenever an established system of doing things is being bypassed.

Naming conventions

Multi-Word names should either have each word separated by an underscore or each word's first letter be capitalized.

Variables should use standard Hungarian notation whenever possible. For custom types, a new Hungarian-like notation should be thought-up and used. This notation should always come before the variable's actual name and should be entirely lowercase. Classes and structs should avoid using 'm_' in their names, because when you really look at it, it's completely unnecessary.

Functions names need only adhere to the Multi-Word naming conventions, parameters should be in Hungarian notation with their purpose somehow included in the name.

Coding guidelines

- Precompute whenever even remotely possible.
- Measures must be taken to prevent copy paste errors.
- Measures must be taken to prevent copy paste errors.
- The STL is evil (most of the time).
- Unnecessarily virtual functions should be rewritten to remove virtual requirements.
- Sloppy code must be marked for recoding later.
- Unreadable code earns you a free punch in the face.
- Measures must be taken to prevent copy paste errors.

Source control

All code modules will be kept in a central CVS repository at DigiPen with a secondary backup location offsite. Backups to the offsite location will be automated on a weekly basis.

Checkouts can be performed at will (network permitting), but all code checked into the primary code branch MUST compile. Anyone caught checking-in uncompilable code will be tarred and feathered.

Memory map

System Memory

The primary systems of the game are allowed to allocate up to the following amounts of memory for internal use.

Entity	Reserved Memory
GDot	16KB
+ Inputskee	
KeyBindings	128KB
Audio	16MB
Timer	0B
GameLogic	4KB
PhysicsMan	8KB
LevelMan	4KB
WidgetMan	4KB
LevelInstance	1MB
AIMan	4KB
TextMan	8KB
+ TextObject	
Sketcher Engine	80KB
Total:	17,664KB

Assuming a giant executable file of 3MB, this brings total system memory usage to ~20MB.

Video Memory

Back Buffer (assuming 640x480 32bit game resolution) 1,228,800B
(1024x768 32bit)
3,145,728B
Portals (assuming 640x480 32bit game resolution) 2,457,600B
(1024x768 32bit)
6,291,456B

Letter textures for the text overlay 1,048,576B

Level Textures:

- Dirt Walls: 6 - 512x512 32bit (3 color, 3 bump) 6,291,456B
- Dirt Floor: 4 – 512x512 32bit (2 color, 2 bump) 4,194,304B
- Dirt Ceiling: 2 – 512x512 32bit (1 color, 1 bump) 2,097,152B
- Concrete: 2 – 512x512 32bit (1 color, 1 bump) 2,097,152B
- Chain Link Fence: 2 – 256x256 32bit (1 color, 1 bump) 524,288B
- Metal Plating: 4 – 256x256 32bit (2 color, 2 bump) 1,048,756B
- Lava: 2 – 256x256 32bit (1 color, 1 bump) 524,288B

Total: 16,777,396B

Particle Textures:

- Steam: 1 – 256x256 32bit 262,144B
- Dust: 1 – 256x256 32bit 262,144B
- Demon Fireball: 1 – 256x256 32bit 262,144B
- Torch Fire: 1 – 256x256 32bit 262,144B

Total 1,048,756B

Character Textures:

- No-Knees: 1 – 512x512 32bit 1,048,756B
- Wally: 1 – 256x256 32bit 262,144B
- Demon: 1 – 512x512 32bit 1,048,756B
- Lava Turtle: 1 – 256x256 32bit 262,144B
- Impy: 1 – 512x512 32bit 1,048,756B

Total: 3,670,556B

Miscellaneous Object Textures:

- Metal Crate: 2 – 256x256 32bit (1 color, 1 bump) 524,288B
- Button: 2 – 256x256 32bit (1 color, 1 bump) 524,288B
- Boulder: 2 – 256x256 32bit (1 color, 1 bump) 524,288B
- Torch: 2 – 128x128 32bit (1 color, 1 bump) 131,072B

Total: 1,703,936B

Total:

- Assuming 640x480 32bit resolution: 27,935,620 Bytes
- Assuming 1024x768 32bit resolution: 33,686,404 Bytes

User Interface

Game Menus

Startup Menu:

The startup menu will consist of a 2D menu with the following choices:

- "Play Game" – This opens the Game Mode Menu.
- "Options" – This opens the Options Menu.
- "Showcase" – This loads a room designed to show off as much as artist work as possible.
- "Quit" – Quits the game and returns to the Windows desktop.

Game Mode Menu:

The game menu will also be 2D with the following choices:

- Easy – Play the game with game play as described in the GDD.
- Impy – Similar to Easy mode, but Impy's in every level and tries to hinder you.

Once either choice is made, the game loads the level selection room.

Level Selection Room:

This is a room that acts as a 3D menu. The room has several closed metal doors on the wall. There's a door for each room that the player has been to. Each is labeled with the name of the puzzle that it leads to, and has a button in front of it. The buttons open their respective door when pushed. Only one door can be open at a time, so opening a door will shut any previously opened doors. These doors connect directly with the entrance of that room, so it's a seamless transition into the game.

The puzzles in the game come in a progressive order, so once a player has beaten a puzzle, they will be exiting one room and going directly to the next puzzle. There is no need to come back to this menu to go from level to level unless the player wants to go backwards or skip to later level which they have already earned.

Options Menu:

The options menu is your average options menu with 3 sub-menus, one for video, another for audio, and a third for game controls. 'Video' will control full-screen toggles, resolutions, vsync, etc…. 'Audio' will control volume levels. 'Game controls' will allow for easy key binding and mouse sensitivity/inversion.

In-Game Controls

Console:

At all times during in-game play, the player will have access to a drop-down console similar to that in Half-Life and the Quake series. This system will allow the player to execute any command that is available to users. Such commands are for binding keys, taking screenshots, quitting the game. Any console command is also bind-able to keyboard keys for quick access during gameplay. See appendix "Console & Key Binding" for detailed information on the inner workings of the console and key binding systems.

Movement:

Movement will take place with a standard 4-command strafing setup. One command will walk forward, another backward, one for strafing left, and the last will strafe right. These commands can be bound to whatever keys the user desires but will be bound to all standard sets by default. These standard sets are:

- W/S/A/D
- NUM8/NUM2/NUM4/NUM6
- UPARROW/DOWNARROW/LEFTARROW/RIGHTARROW.

Looking Around:

The game is designed to use mouse free-look. But for the oddball players, keyboard support will be provided for rotating as well as looking up and down.

Camera controls will appear in two forms. One command will be bind-able for cycling through the 3 camera modes (detailed in the video section). Three additional commands will be bind-able for quickly switching to a specific mode.

Portal Summoning:

The game is designed to have one portal be opened by left clicking the mouse and the other by right clicking. For the oddball players, keyboard support will be provided.

In-Game Menu:

The in-game menu can be brought up with a command that will be default bound to the escape key. The menu will pause the game and bring up the following options:

- "Return to Game" – Un-pause the game and continue playing.
- "Restart Room" – Restarts the current level. A confirm screen may be used depending on focus group results.
- “Options” – Loads up the same options menu specified in the “Game Menus” section above. The player can change video/audio settings and rebind their controls.
- "Quit to Main Menu" – Takes the player back to the main menu. A confirm screen will definitely appear.
- "Quit Game" – Quits back to Windows desktop. A confirm screen will definitely appear.

Death Menu:

Narbacular Drop has no concept of there being a number of lives. Each time a player dies they will be instantly respawned at the level’s specified spawn point. This eliminates all need for menu gibberish. It also means that there’s no way to lose in Narbacular Drop, the worst thing that can happen is that the player can quit.

Win Screen:

When the player beats Demon, something will happen to acknowledge this triumph. This will probably be a FMV followed by getting sent back to the main menu.

Graphics

We will be using a Direct3D based graphics engine built in-house that has been dubbed as the "Sketcher Engine". The engine abstracts the process of rendering to a managed but still extremely flexible level. All Direct3D configuration settings, management, and initializations are done through simplified engine calls.

Features

- Automatically finds the fastest possible configuration for any given display mode.
- Automatically enumerates all video adapters and their capabilities.
- Stores current capabilities in easy-to-reference places.
- Built-in fog manager.
- Built-in light manager with automatic configuration of active lights to keep the active light numbers within the video card's supported limit.
- Built-in texture allocation manager with normal based bump map generation support.
- Built-in 2D text overlay for quick debugging output.
- Built-in world matrix stack.
- Built-in camera controls with multiple methods of specifying position and direction.
- Built-in support for wireframe toggling.
- Easy control over display properties with automatic "closest-match" finding.
- Automatic surface loss handling.
- Built-in culling frustum for dramatically improved performance.
- Built-in allocation and management for static and bone-based animating meshes from "X" files.
- Complete flexibility of renderable objects. (See appendix "Sketcher Engine")

For details on the inner workings of the Sketcher Engine, please consult appendix "Sketcher Engine"

Formats

Textures

Texture loading is done through the D3DX library; so all common formats are supported (formats that work in Internet Explorer).

Alpha-blended textures should be in D3DX's native DDS format. Non-blended, but alpha-masked textures should be in any valid BMP format. Paletted run-length-encoded bitmaps would be the most space saving but will not be required. Non-blended, non-masked textures should use full quality jpeg compression.

Bump maps should be full quality gray scaled jpegs.

Static and Animating Meshes

All meshes must be in Direct3D's "X" format. All animation must be done through the use of bones and mesh skinning.

In the event that a mesh cannot be animated with bones, each keyframe of animation must be a separate "X" file; the game will interpolate each vertex from one keyframe to another.

Vertex and Pixel Shaders

Vertex and pixel shaders should be created as standard ".fx" files. Specific shading functions can be extracted from compiled ".fx" files after compilation by Direct3D.

View Modes

Three viewing modes will be supported:

1. First person.
2. Third person follow-cam.
3. Third person static camera positions specified by level.

Portal Effect

The creation of our portals takes place with some extremely simple trickery along with a lot of tedious things to make it work well.

Essentially, a portal is just a textured quad. But the contents of the texture are dynamic and the texture coordinates change based on how you're looking at the quad.

To create the contents of the texture:

1. Convert the current camera position and look vector into coordinates relative to the portal.
2. Find this position and rotation in relation to the portal's exit.
3. Move the camera to the point and rotation found in step 2.
4. Render the scene while culling all objects between the camera and the portal exit.
5. The render surface is now the source texture for the portal and the camera should be moved back to where it was before step 1.

To render the quad with the proper texture coordinates:

1. Convert each coordinate into screen space.
2. Copy the normalized screen space x and y components into the u and v coordinates of each vertex.
3. Render the quad.

Bump Mapping

Bump mapping is being implemented through the use of vertex shader v1.1 and pixel shader v1.1 capabilities. The vertex shader is responsible for capturing the specified light position, calculating the vector from the light to the current vertex, and then to smuggle that vector to the pixel shader. The pixel shader is responsible for reading the bump map vector from the bump map itself, applying a dot product against the interpolated light vector, and applying highlights to the regular texture based on that dot product.

Audio

The audio engine is a wrapper that simplifies and enhances the most useful elements of the *FMOD engine*. FMOD is an extremely powerful and stable sound engine library that is built on top of DirectSound. It is free for non-commercial use.

"Yes that's right, if your product is not intended to make any money, and is not charged for in any way, then you may use FMOD in it for FREE!"
–The FMOD Website.

Features

The following features are available through the audio engine wrapper:

- 2D/3D sound effects.
- 2D music with support for compressed file formats.
- Fine control over volume, playback speed, and playback point.
- Music "stacking", which enhances transitions.
- Easy randomization of repetitive sound effects.

For details on the inner workings of the audio engine, see appendix "Audio Engine".

Formats

FMOD supports virtually any type of audio format most people have ever heard of. Here's a quick rundown of it's most popular supported formats: MOD/S3M/XM, MIDI, WAV using any ACM codec, ADPCM, Ogg-Vorbis, MP2/MP3, WMA/ASF, AIFF, CDDA, as well as a plethora of internet stream types.

Any of these formats can be used for either music or sound effects. But for efficiency, usage of heavily compressed formats will not be used for quick sound effects. For space savings, music should make use of a "lossy" compression such as MP3, Ogg-Vorbis, or WMA.

The current recommendation is WAV format for sound effects and Ogg-Vorbis for music. But as stated above, this standard is flexible.

Task List

The following is an unordered list of tasks and whom they're assigned to.

Game Object Tasks:

- **No-knees**
  - Player Controls (Garret)
    - Up
    - Down
    - Left
    - Right
    - Portal A
    - Portal B
    - Changing camera Mode
    - Mouse look (for first person)
  - Switching animations: State Machine (Garret)
    - Idle
    - Walk
    - Run
    - Strafe
    - Hitting an object
    - Portal Summoning
      - Move the head to match the placement of a portal.
    - Knocked Down
    - Falling
    - Alignment while falling through portals
    - Getting up from the ground
    - Death…
  - Death Conditions (Garret)
    - Restart Level
    - Box/Boulder on the head
    - Falling in lava
    - Fireball to the face
  - Sound Effects (Jeep creates) (Garret adds state)
    - Footsteps (metal, ground, chain link)
    - Falling
    - Hitting ground
    - Running into objects (chain link, wall)
    - Death
- **Wally**
  - AI (Kim)
    - Context sensitive scripted events for help.
    - Beginning scripted event for basic portal tutorial.
  - Write all scripted events so they may be passed to Jeep for synthesis.
  - Synthesized Wally Voice (Jeep)
  - Sound Effects (Jeep)
    - Portal open/close
    - Portal rejection
- **Impy**
  - AI (Kim)
    - Acknowledge all walls and objects in the room.
    - Know when a box has been moved, and bring it back

- Path finding algorithms
    - Navigate around lava
    - Navigate around edges, and know when to climb
- Player chase algorithm
- Fleeing algorithm
- States in a state machine for Impy:
    - Wander – Walking
    - Running – Goal
        - Running toward a box
        - Running toward the player
        - Running away from player
    - Climbing – Walls
    - Knocked Down
    - Tantrum
    - Blocking portals
    - Clean
        - Moving boxes
        - Idle Animation

- Sound Effects (Jeep)
    - Climbing (chain link, wall)
    - Footsteps (running, walking)
    - Lifting (heaving noise)
    - Cackling
    - Tantrum noise
    - Cleaning (sweeping, polishing)

- **Demon**
    - AI (Kim)
        - Path finding algorithms around each level of the boss battle.
        - States in a state machine for:
            - Fireball Shooting
            - Summoning Boulder
            - Jump
            - Wandering
            - Gloating
            - Getting Hurt
            - Dying
    - Sound Effects (Jeep)
        - Summoning
        - Getting hit by fireball
        - Roaring
        - Jump
        - Death
- **Lava Turtle**
    - AI (Kim)
        - State Machine
            - Wander
                - Follows a path between points.
            - Idle
                - Stranded on land
                - Random while in lava.
    - Sound Effects (Jeep)
        - Idle noise.
        - When landed on.

        - When turtle lands in lava.
    - Collision with the player, so they can be used as vehicles.
- **Boulder (Jeep and Garret)**
    - Attach physics
    - Sound Effects
        - Rolling
        - Boulder hitting boulder
        - Boulder hitting the ground
- **Mesh Fence (Jeep)**
    - Sound Effects
        - See Player and Impy
        - Box hitting mesh
        - Boulder hitting mesh
- **Boxes (Jeep)**
    - Attach physics
    - Sound Effects
        - Sliding
        - Hitting ground
- **Fireball (Jeep)**
    - They go straight
    - Particle System
        - Flame trail
    - When to initialize and de-initialize
        - Disappear when they hit an object/wall
    - Sound Effects
- **Buttons (Garret)**
    - Detecting button depression, coding a collisions list (boxes, boulders, player, etc)
    - Sound Effects
        - Button up/down
- **Lava (Jeep) (follows particle system)**
    - Rigid Body motion
    - Collision with objects, especially the player
    - Falling through portals
    - Hardening
    - Sound Effects
        - Bubbling noise
        - Lava flowing through a portal.
        - Lava solidifying.
- **Camera (Garret)**
    - AI
        - Movement when the player is close to the wall, portal, and forcing to first person
        - Fixed camera movement to keep player in view
    - 1st Person Implementation
        - mouse input : move camera left, right, up, down
        - constrain perspective
        - mouse wheel scroll to third person
    - 3rd Person Implementation
        - mouse input
        - keep camera a specific distance from Noknees
    - Fixed
        - Determine camera locations per room.
        - Keeps the player in view
- **Portals (Dave)**

    - AI
        - Locked portal movement around corners, etc. (low priority)
        - Implement playing of Wally animation opening/closing
    - Graphical overlay for positioning of portals (sorta like a crosshairs), an image of Wally with a black mouth, either alpha blended red or blue, or gray if you can't place a portal.
    - Sound Effects
        - Refer to Wally
    - Collision (Jeep)
        - Need to annul collision when an object is passing through a portal
        - What surfaces a portal can stick to.
    - Visual
        - Conditions to prevent infinitely seeing a portal.
- **Torches (Jeep)**
    - Their lighting system
    - Getting them to flicker
    - Particle system for the flames
    - Sound Effects
        - Ambient flickering noise.
- **Doors (Jeep)**
    - Sound Effects
        - Door opening chime

Level Tasks:

- **Level 1: Polishing Up**
    - Create level in WorldCraft (Garret)
    - Create and register AI map. (Kim)
    - Create AI goals for Impy (Kim)
    - Wally hint contexts (Kim)
        - Still in cage after a long section of time
        - Upon exiting cage, talk about buttons.
        - After three minutes w/ no boxes on buttons, suggest to use a box.
    - Win conditions for the level
        - Escape cage
        - Three boxes on buttons
        - Exit through door
- **Level 2: Boulder Dash**
    - Create level in WorldCraft
    - Create and register AI map.
    - Create AI goals for Impy
    - Wally hint contexts
        - Comment on the turtle being out of lava
        - If the player hasn't gotten past the lava (in 3 minutes) or has died in the lava, suggest something about the turtle.
        - If the player has reached the button-side, and the door hasn't opened in 3 minutes, suggest something about boulders on the buttons.
    - Win conditions for the level
        - Cross both boulders
        - Cross lava
        - Press both buttons
        - Exit room
- **Level 3 : Hallway To Hell**
    - Create level in WorldCraft
    - Create and register AI map.
    - Create AI goals for Impy
    - Wally hint contexts

        - Suggest lava turtle for first pit
        - Suggest using pillars for second pit
        - Suggest using portals on the pillars for the third pit
        - Suggest using the last pillar to get to the door.
    - Win conditions for the level
        - Cross first lava pit
        - Cross second lava pit
        - Cross third lava pit
        - Land on exit platform
        - Exit level
- **Level 4: "Beware of Falling Rocks"**
    - Create level in WorldCraft
    - Create and register AI map.
    - Create AI goals for Impy
    - Wally hint contexts
        - When on the same level as the button
    - Win conditions for the level
        - Stop the boulders from spawning
        - Get to the top ledge, and exit the level.
- **Final Level: "Fire With Fire"**
    - Create level in WorldCraft
    - Create and register AI map.
    - Create AI goals for Demon
    - Wally hint contexts
        - Every few minutes cheer on NoKnees
        - Tell NoKnees to turn Demon's weapons against him
    - Win conditions for the level
        - Hit Demon on level 1
        - Hit Demon on level 2
        - Hit Demon on level 3
        - Hit Demon on level 4
        - End of the game!!! YAY!!
- **Artist Room** (Done this semester) (Garret)
    - Create level in WorldCraft
        - Pedestals:
            - Lava Turtle
            - Demon
            - No-Knees
            - Impy
        - Shelf
            - Boulder
            - Box
        - Doors
            - Door in/out to the menu
        - Wally on a wall
        - Pit of lava in the middle… covered by a grate??

Graphics/General Engine:

- Bump Mapping (Dave)
- Displaying Animated Meshes (Dave)
- Volume Textures (Dave… second semester, if particle systems don't work)
- Loading levels (Jeep)
    - Pretty much done
- Particle System (Dave)

    - Lava Steam
    - Fireballs
    - Dust
- Lighting Engine Revision (Dave)
    - Torch flickering
- In-Game Text (besides console) (Jeep)
- Overhaul Inputskee
    - If we fail to acquire keyboard and mouse input, keep trying to get it.
        - Causes game to crash…

Tools:

- Artist's Application (Dave) when done w/ TDD
    - Project that allows artists to view their models, and textures.

Physics:

- Newtonian Physics (done)
- Rigid Body for lava (Jeep)
- Rotational Inertia for objects (Kim)

AI: (This is the realm of Kim (except the Camera Object))

- Camera AI
    - View Camera Object
- Player AI
    - Refer to No-Knees object
- Impy AI
    - Refer to Impy Object
- Demon AI
    - Refer to Demon Object
- Lava Turtle AI
    - Refer to Lava Turtle Object

User Interface/Various Stuff that no one wants to deal with:

- Menus (All Dave) Towards the end of this semester….
    - 3D Selection Room
        - Create rooms in WorldCraft
    - Pause Game Menu
        - Text Configuration of Menus
        - Implementation of the following menu tasks:
            - Return to Game
            - Restart Room
            - Quit to Main Menu
            - Quit Game
    - Options Menu
        - Text Configuration of Menus
        - Implementation of the following menu tasks:
            - Resolution (1024x768, 800x600, 640x480)
            - Anti-Aliasing (on/off)
            - Invert Mouse (on/off)
            - Sound Effects Volume (+/-)
            - Music Volume (+/-)
            - Hints (on/off)
- Saving the game (auto save, how many levels unlocked saved)
    - Need to implement the system to collect information needed for a saved game.

- Game Installer (Dave)(next semester)
- Intro Screens (Kim next semester)
    - DigiPen Logo
    - Nuclear Monkey Software Logo
- Credits Sequence (Spiffy McGee)

Sounds: (Domain of Jeep)

- Sound Effects
    - View objects
    - Ambient dungeon noises??
- Music
    - Music for menus
    - Music for dungeon
    - Music for boss fight
    - Music for the credits

Product Management

- Box Cover (Jeep)
- Manual

Timeline / Milestones

Milestone Schedule

- 11/19/04 – Engine Proof: Worldcraft level loader, Camera system, Basic physics, Animated meshes. Portals partially implemented.
- 12/10/04 – First Playable: Box, button and boulder game objects implemented. Portals fully implemented.
- 02/08/05 – Pre-alpha: All A-Level features functionally working.
- 03/15/05 – Alpha: All A-Level features complete and debugged.
- 04/06/05 – Beta: All B-Level features functionally working.
- 04/20/05 – Gold: All B-Level features complete. Game crash-proofed.

Timeline

Please refer to our project planning timeline.

Installer

For the actual installation of files, we will be using "Inno Setup" developed by jrsoftware (http://www.jrsoftware.org/isinfo.php).

Wrapped around the installer will be our autorun application, which will be custom-made for this game by the Nuclear Monkey Software team. This autorun will run in two possible modes. One will be flashy for high-end systems, the other will run only simple graphics for low-end systems.

The autorun application will have the following options:

- "Install" if the game has not been installed. "Play" if it has been installed.
- "Install DirectX", which will install the DirectX runtimes necessary to play the game.
- "View Readme", which will open the game's release notes in the OS's default text editor.
- "Nuclear Monkey Software on the Web", which will open the Nuclear Monkey Software website in the OS's default web browser.

Appendices

Console & Key Binding

Purpose: To give players fine control over their input scheme.
Maintainer: Jeep Barnett.

Console:

The Console is designed to be a flexible debugging and scripting tool. Unlike most entities it remembers info between being destructed and created. When it is created it switches the input stream to 0x1337BEEF and displays info. It can now receive text input. When it is destroyed it does not get text input or display anything, but commands can still be called explicitly. Pressing Tilde (~) toggles the console on and off.

RunCommandScript is for passing in unparsed text as it would normally be typed. ProcessCommand is used if you know the specific index of the macro, but haven't parsed it for parameters yet. CallMacro is used to call a specific macro with a list of pre-parsed string parameters. Don't worry, you'll probably never have to call these functions... they exist to be used by the binding system.

Manually entering text is handled internally by the console. All standard keyboard keys and the shift key should be processed normally. Pressing Return calls RunCommandScript on the current input string. Commands are not case sensitive and large whitespace blocks are ignored.

Commands are integrated into the console. Users can simply type the macro and a list of parameters for that command separated by spaces. The simplest command “Help”. This gives you a list of all other commands. Typing “Help” and the name of a command will give more information about the command.

To make a new command complete the following steps:
1) Add a new macro define to the MACRO enumeration in ConsoleMacros.h.
2) Define what the macro does in ConsoleMacros.cpp.
3) See other macros for examples of how to parse the parameters and handle errors.
4) Add info about the macro where MACRO_HELP is handled at the top of ConsoleMacros.cpp.
5) Make a string command that is tied to the macro in SetCommands in Console.cpp.

When the console is first initialized it will look for a file called OnStartScript.txt in the root directory. Each line of this file will be run as a command. This is an easy way to call commands that set defaults (and options) for the game.

Bindings:

The purpose of bindings is to make single keystrokes activate console commands. AddKeyBind is used to bind any key to a command. It's a complicated function to use and is really just used as an interface for the console's Bind macro. The parameters for the bind macro are the name of the key, the input stream, the command, and the parameters it calls on the console. An input stream of -1 means any input stream. Putting a + after the name of the key binds it as “WhenTriggered” rather than “WhenPressed”. EX: “Bind F11+ -1 ToggleFPS” will bind the F11 key to toggle the FPS display when it's triggered on any input stream. When Binding's think is called it will call the macros of any activated binding.

As mentioned in the “User Interface” section, at all points in the game, the player will have access to an options menu which will contain a controls menu. This controls menu will allow for easy rebinding of keys to the most common commands.

Audio Engine

Purpose: To create a robust and simple-to-use audio system.
Maintainer: Jeep Barnett.

The FMOD library and include files need to be added to the include directories of the project. Fmod.dll also needs to be included in the root directory of the project and builds. All necessary files can be downloaded from http://www.fmod.org.

The main feature of the wrapper is that it defines 2D background music and 2D/3D sound effects as separate objects, while in FMOD they are the same object with different parameters. By choosing to play as sound effect or music it automatically configures these parameters for maximum simplicity and efficiency. It also manages separate global volume controls for music and sound effects.

Playing a *sound effect* is fairly basic. Call the PlaySFX and pass the name of the file to play. It will play it as a 2D sound at maximum volume. It returns a reference handle so that it's parameters (volume, pan, pitch, etc.) can be modified dynamically with simple wrapper functions. This reference can also be used directly with FMOD's functions for complete flexibility. You may also choose to initialize and pass a SFX_Init struct to PlaySFX to set its parameters at creation.

PlayRandomSFX can take an infinite number file name parameters and randomly chooses one of them to be played (for less repetitive footsteps, etc.).

To play a *3D sound effect*, the b3D bool in the SFX_Init struct must be set to true. Then the SFX3D_Init member of the struct can be initialized with important 3D data (position, velocity, emission type, etc.). The returned reference can also be used to dynamically change 3D parameters. A *global reverb ambiance* can be set that affects all 3D SFX (so long as the machine has a sound card that supports EAX).

2D music is played with the PlayMusic function. This is designed to assume that you usually only want one music track playing at a time. So calling PlayMusic when a track is already playing will stop the previous track and start the new one.

However, parameters can be set that allow several tracks to be played at once or for the tracks to be *stacked*. Stacking a track means that you choose the amount of seconds that it will take for the current track to fade to mute and the amount of seconds that it will take for the new track to ramp to full volume. An infinite number of tracks can be stacked (capped by a variable maximum) and all previous tracks that have faded continue to play (silently). When StopMusic is called, the most recent track will fade and the track before it will ramp to full volume. StopAllMusic will stop all the stacked music tracks.

For *looping positional music and ambiance*, 3D sound effects should be used with bLooping set to true.

Simple *MIDI* support is also wrapped (play, stop, pause), but serious use of them should be done using the FMOD interfaces.

Sketcher Engine

Purpose: Graphics abstraction and management.
Maintainer: Dave Kircher

Overview:

The Sketcher Engine is designed to abstract all Direct3D requirements into automated systems that can be controlled through easy to understand functions. Also, its secondary purpose is to provide heavily optimized management for common graphics resources such as textures, lights, static/animating meshes, text to screen, and fog.

The code itself is designed to take advantage of namespace pop-ups within most C++ IDE's to help coders find the function/variable they need. It does this by separating the engine into 2 portions, both existing as a single instance in global scope.

The viewable/public portion of the engine is a class comprised of subclasses for each logical division of functionality (camera, textures, fog, lighting….) in a header. Each class and subclass contains ONLY static functions and static variables. So, if someone were to create an instance of the Sketcher Engine, it would contain nothing. As described before, these nested classes allow coders to see all functions and variables available in a section through the namespace pop-ups.

The hidden internal portion of the engine is a set of namespaces nested in a similar fashion as the public class. This namespace is kept within the engine's cpp file, so it's guaranteed to be hidden from the world. This hidden set of namespaces contains internal data management information such as arrays of allocated textures and lights.

The last portion of the engine consists of variables that are made read-only in the public static classes. These are variables such as the current forward vector in relation to the camera, width of the screen, display format, things of that nature, which are good to know, but should only be changed by the engine itself. These variables are made read-only by first being created as a normal variable within the private namespaces. Then a constant reference to the private variable is made in the public classes, which allows the variable to be changed by the engine, and allows coders to read the variable, but not change it without an explicit recast, which prevents accidental modification. Since the public variable is a reference, there is zero extra memory usage and zero extra processing time to access the contents of the variable.

Public Data Outline:

The primary class is “Sketcher_Engine” defined in “Sketcher_Engine.h”.

Subclasses:

- D3D – COM interfaces to Direct3D and the current Direct3D Device.
- Initialization – Functions that are only needed during initialization and which cannot be used at any other time.
- Uninitialization – Functions that need to be used to shut down the engine and clean up used memory.
- GeneralConfiguration – Functions for changing the display setup (width, height, full-screen/windowed, format….). As well as several Booleans to control generic rendering effects such as wireframe mode.
- Matrices – Current world, view, and projection matrices. Also contains a set of functions to implement a world matrix stack.
- Camera – Variables to control camera position and rotation. Functions to control far viewing plane, field of view, as well as functions to look at a point in space or match a look vector.
- Screen – Mostly read-only variables for screen width, height, format, and whether it’s full-screen or not. Also contains functions for taking screenshots and converting from normalized 2D coordinates to screen pixel coordinates and vice-versa. Lastly, contains the variable and function to control the screen’s clearing color.
- Scene – Functions for adding/removing objects from a scene, and rendering a scene.
- TextOverlay – The text overlay simulates a 2D array of text that will be put on the screen every frame that the overlay is set to visible. This class contains functions to control number of rows and columns in the array, a Boolean to control visibility, and functions to get either row or cell pointers quickly. Text can be placed reliably within this array with simple memory copies such as sprintf() and the screen will reflect these changes automatically.
- SurfaceLoss – Contains 1 function that needs to be called when a surface is lost. The engine will internally detect surface loss when trying to render a frame and will call this function automatically. Also contains 2 callback function pointers to be set by the coder if they need to free and restore any custom resources in the case of surface loss.
- Fog – Variables to control fog density, mode, color, and near plane. The fog’s far plane is automatically set to be the same as the camera’s far plane (so that objects past the far plane will be culled by the view frustum instead of rendering all pixels as the fog color).
- Textures – Functions for managing texture allocation. Getting a texture or bump map is done by filename (string lookup is optimized via string trees). Freeing a texture or bump map is done by Direct3D texture pointer. Lastly, the set of textures used by the text overlay is made publicly available through a read-only array of 256 textures with each slot in the array containing a texture representing the ascii value of its index. This allows for extremely easy mapping of text to texture.
- Lighting – One variable, which allows direct control over the ambient lighting color. One read-only variable reporting the maximum number of active lights supported by the video card. Two functions for creating and destroying lights. Creation returns a D3DLIGHT9 pointer, the pointed structure can be changed by the coder at will and any changes will automatically take effect upon rendering.
- Capabilities – Read-only data specifying the capabilities and supported display modes of every display adapter in the system.

Usage Notes:

Creating a renderable object:

The Sketcher Engine does not define any internal types of geometry or drawing methods except for its own text overlay. Rendering of objects is achieved through a system of callbacks wrapped within a class for ease of understanding.

When a coder wants to create a renderable object. They must create a class that inherits from "Sketcher_Object_Base" which is defined in "Sketcher_Object_Base.h". This base class is an abstract base class with several variables that are used to test the object, no matter how abstract, against the viewing frustum for quick culling. These variables are:

- vCenter - D3DXVECTOR3 used to define the object's center point.
- fRadius - which defines the distance from the object's center to it's furthest vertex, this value can be a guesstimate if needed at the cost of culling accuracy.
- bOptimizeMe – Boolean that determines whether the culling test should even be run on the object. Setting this to false means that the object will automatically pass all viewing frustum tests.
- bInvisible – Boolean to allow for automatic non-rendering of the object. If this is true, then the object will not be drawn.

In addition to the above variables, the base class contains a static constant reference to Sketcher_Engine's copy of the Direct3D Device pointer with the name pD3DDevice. This allows all derived object to refer to the commonly used Direct3D Device pointer more easily.

Lastly, the base class contains 1 pure virtual function called "GetClassInformation" that must be re-implemented by all deriving classes. The function takes a reference to a "SO_ClassInfo" structure, which the function is supposed to fill in with relevant data. This function will only be called when the object has been added to the scene and the engine has never encountered an object of its type yet. The engine determines differences in object types by reading and storing vftable pointers from objects, which are guaranteed to be unique for each class so long as it's filled in the GetClassInformation function, and even reliably unique in cases where the function is not filled in by the derived class (derived from derived classes that filled it in).

The SO_ClassInfo struct has the following variables:

- pfn_Draw – A class specific draw function must be cast to type "SO_FuncPtr" and stored here. The function stored here must return void and take void. This function will be called whenever the engine is rendering a scene and the object has passed all culling tests and needs to add it's data to the screen. Within the draw function, the object is allowed to assume for all intents and purposes that it is the only object being drawn and can modify render states and draw primitives as such. The object need only be considerate to other objects of its same type when changing render states in the draw function. Render states are reset between object types, but not between objects of the same type.
- pfn_AllocateDrawingResources – Another class specific callback that is of type "SO_FuncPtr" (takes void, returns void). This should be a callback to a function that will allocate Direct3D resources such as vertex buffers and textures that are required to draw the current object. The engine will call the function under 2 circumstances: The first is when the object is first added to the scene. The second is when the Direct3D Device had to be reset for some reason and the object should re-allocate its resources.
- pfn_FreeDrawingResources – Another class specific callback of type "SO_FuncPtr". This should be a callback to a function that will free ALL Direct3D resources used by the current object. The engine will call the function under 3 circumstances: First, the object has been removed from the scene by code in the driving application. Second, the Direct3D Device is about to be reset for some reason (which requires all resources to be freed first). Third, the engine's uninitialize function has been called and the object was still in a scene list.
- StateBlock – This is of type IDirect3DStateBlock9* and should be filled with a state block interface. The contents of the state block should be render state settings that are necessary to draw

every conceivable object of this class type. The contents of this state block will be applied after all renderstates have been reset to the engine defaults but before any object of this class type is rendered. This application is only performed once per class type. The class should account for any render state changes that take place within a drawing function of this class as well. Proper setup of a good state block is key for drawing efficiency. The intended method for filling in this variable is to first call BeginStateBlock() on pD3DDevice. Then to set the render states as normal. Lastly calling EndStateBlock(&sci.StateBlock) on pD3DDevice, which will fill in the variable with changes you have just made and no others. It should be noted that the engine itself is responsible for releasing state blocks, so the class does not need to keep track of its state block in any fashion. DEFAULT: NULL

- bNeedsLighting – This Boolean should be set to true if the object class intends to be lit (even if only in some cases) and wants to cooperate with Sketcher_Engine's built-in light manager. If this is set to false and the object has set the D3DRS_LIGHTING renderstate to true, then lighting behavior is undefined unless strictly managed by the class itself. DEFAULT: false
- bUsesAlphaBlending – This should be set to true if the object does alpha blending. This variable is KEY to good alpha blending. If set to true, this class of object will be drawn after all classes of objects that do not use alpha blending. The objects within the class will also be sorted from back to front. It should be noted that color key alpha masking does NOT count as alpha blending. Color keys do not cause rendering difficulties and do not need the special processing that this Boolean entails. DEFAULT: false
- GeneralComplexity – This unsigned char is intended for insane levels of scene optimization by the engine. An object should set this value somewhere in the range of 0 to 255 depending on the complexity of the objects it draws. 0 is intended for objects of few polygons and no lighting. 255 is intended for objects like lit animating meshes. In the case where the complexity cannot even be guesstimated, a value of 127 should be used so that no biasing in either direction occurs. DEFAULT: 127
- GeneralClosenessToCamera – Another unsigned char like complexity above. 0 is intended for HUD objects. 255 is intended for skyboxes. It is understood that many objects move in the world, so this is just a general guesstimate. DEFAULT: 127.

GeneralComplexity and GeneralClosenessToCamera are used to determine object class drawing order in non-alpha-blended objects. Although alpha-blended objects are not guaranteed to take these values into account, they should still be filled in for profiling reasons.

Example Usage:

```
Sketcher_Engine::Initialization::Initialize();
Sketcher_Engine::Initialization::TakeControl( hWND );
        //where hWND is a handle to a window you've created.

Sketcher_Object_Foo bar;
        //where Sketcher_Object_Foo is a class you define that derives
                //from Sketcher_Object_Base.

Sketcher_Engine::Scene::AddObjectToScene( &bar );
while( bSomeLoopBoolean)
{
        Sketcher_Engine::Scene::RenderScene();
}
//optionally Sketcher_Engine::Scene::RemoveObjectFromScene( &bar);
        //if you're paranoid

Sketcher_Engine::Uninitialization::Uninitialize();
```

WorldCraft

Worldcraft has been the standard in editing levels for games using Id Software's engines (Doom, Quake, Half-Life) for over a decade. The outstanding feature is that classes of objects and new data structures can easily be created and placed in Worldcraft. This feature allows Worldcraft to be used for nearly any game so long as the classes are customized.

Classes are customized in the FGD files. There's 3 types of entity classes: base, solid, and point. Base classes are designed to be derived from and cannot be instantiated. Solid classes are for entities that are bound to geometry created within Worldcraft. They are useful for moving platforms, buttons, etc. Point classes use no geometry from Worldcraft and exist at a single point. They are useful for light positions, respawn points, etc. For detailed descriptions of the syntax for creating an FGD file, please see WorldCraft.pdf.

Worldcraft uses WAD files to store all of its textures. These can easily be created using a program called Wally. Simply import the textures that you want into a single WAD. Keep in mind that WAD files hold 256 color bitmaps, so it's encouraged that you keep a separate folder of the original high quality textures for actual use in the game. Read WAD3Files.pdf for more details.

To create a level in Worldcraft you use the simple set of tools provided. One creates geometry. Another creates point entities. To convert a piece of geometry to an entity right click on it and select “Tie To Entity”. To unbind it from an entity right click and select “Move To World”. Press SHIFT+A to bring up the texture tool. To edit the Worldspawn entity (each level has exactly one Worldspawn) select Map>Map Properties from the title bar menu. That covers the basics, so from here you should play around or find an online tutorial (there's hundreds of good ones out there).

Saving a Worldcraft level creates an RMF. This file is not useful to you. Export the level to a MAP file. This is a text-based format that's syntax is very basic. If you want to parse a MAP file on your own, read MAPFiles.pdf for more details. But I recommend that you use the parser that he has provided. It converts MAP files into CMF files. CMF is a file format that the creator of these documents made and is designed to work very well with custom games as well as optimizing the geometry that Worldcraft creates.

CMF Level Parser:

The public function LevelInstance::Load is used to load a cmf file. This is generally called from the LevelManager. Load first parses the header.

Header:

3 char	CMF ID ("CMF")
1 char	Version byte
1 uint	Number of WAD files
1 uint	Number of entities
1 uint	Number of textures
x char	WAD filenames (zero terminated)
x char	Texture names (zero terminated)

Then it parses through the total number of entities where each entity is like so.

Entity:

x char	Entity class (zero terminated)
1 uint	Number of properties
x Property	Entities properties
1 uint	Number of polygons

	x Polygon	Polygons

The Property and Polygon data structure is below.

Property:

x char	Property name (zero terminated)
x char	Property value (zero terminated)

Polygon:

1 uint	Texture ID
1 Plane	Polygon plane
1 uint	Number of vertices
x Vertex	Vertices

Vertex and Plane are like this.

Vertex:

1 Vector3	Coordinate
2 double	u and v texture coords

Plane:

1 Vector3	Normal
1 double	D

And Vector3.

Vector3:

3 double	x, y, and z component

Using loops within loops and some dynamic arrays we store all this information. We create actual level objects based on this information. First I switch statement the entity class name to figure out which type of object to load. Then each class type has a separate creation function (to keep things clean). The properties for the object are not guaranteed to be in any specific order (or even exist). So one must loop through the total number of properties and compare the Property Names against what names you'd expect to find. When a match is found, its Property Value can be converted from a string to whatever format you need. Polygon data can then be loaded into whatever structure is necessary. Finally using the parsed property data and polygons we can initialize the level object itself.

Level Objects:

Any level object derives from the LevelObject base class. Each LevelObject has a TargetName and a Target that default to null. These are used in the general targeting system. ActivateTarget calls the WhenTargeted function on any LevelObject that's TargetName is the same as the Target of an object. WhenTargeted is used for a LevelObject to do anything it feels necessary when another LevelObject targets it. For example a light WhenTargeted might toggle the light.

Level Manager:

This is designed to handle the very specific way that levels progress in Narbacular Drop. If there are 2 levels in memory when LoadLevel is called it will destroy the oldest. It also ensures that the start of levels newly loaded attach to the end of the most recent level.

Team Sign-Off

Product Manager
Jeep Barnett _____

Technical Director
Dave Kircher _____

Designer
Garret Rickey _____

Producer
Art Director
Kim Swift _____