Technical Design Document

DigiPen Institute of Technology

November 2nd, 2018

Team Pi Conspiracy

Edward Zerbe
Engine Programmer | Physics Programmer
Jiyun Ruan
Physics Programmer | Tools Programmer
Omkar Patil
Producer | AI Programmer | Level Designer | Audio Programmer
Rohit Tolety
Game Designer | Graphics Programmer | Gameplay Programmer
Sujay Shah
Engine Programmer | Level Designer | UI Programmer
Zoheb Mohammed Hynus
Gameplay Programmer | Level Designer
Varun Premchandran
Technical Director | Graphics Programmer | Level Designer | BGM and Sound Designer
Yen Pham
Volunteering Artist

1. Technical Overview

Engine Summary:

The **PiEngine** is a Component based 3D engine developed by PiConspiracy that stands to be the core platform for the development of our games and their prototypes. A 3D engine at its core the Engine is written in C++ and utilizes SDL for Window creation and Input handling and OpenGL for hardware-accelerated graphics. It is data driven and supports 3D audio with the use of the FMOD API.

2. Development Overview

Development Team:

Edward Zerbe: Engine Architecture and Physics Programmer.
Omkar Patil: Producer, Gameplay, AI and Audio Programmer.
Jiyun Ruan: Physics and Tools Programmer
Rohit Tolety: Graphics, Gameplay and Game Designer
Varun Premchandran: Technical Director, Graphics Programmer, Level Designer, Sound and Background Music.
Zoheb Mohammed Hynus: Gameplay and AI Programmer.

Technical Specifications:

Platform:	Windows, 32-bit and 64 bits.
Language:	*C++*
Dev Environment:	Visual Studio 2017
Compiler:	Microsoft C/C++ Optimizing Compiler Version 19.11.25507.1 for x86
Version Control:	GIT, hosted by DigiPen
Coding Style:	https://tinyurl.com/yadtjqtt

Content Creation Tools

- Maya
- Photoshop
- GIMP
- Logic Pro X
- Pro Tools

External Libraries

- SDL2 2.0.9
- GLEW 7.0
- GLM 0.9.9.3
- FMOD 1.10.09
- ImGui 1.66
- ASSIMP 4.1.0
- Nlohmann Json 3.2.0
- STB Image 2.19

- Visual leak detector
- Spdlog 1.2.1

3. Software Development Architecture

The team decided on selecting the Iterative Agile Development Model for our Engine and final Game Design. In accordance with the model, we set weekly goals and made sure they are met at all costs. Once met, we work on improving them and also introducing future goals.

4. Engine Features

Architecture:

PiEngine has component based architecture. Components will impart behaviours to the attached GameObject. GameObject Manager will be responsible to maintain status of the gameObjects. GameObjects in turn will be responsible to keep their components updated. This makes the whole series of updation easy but it comes with a minor drawback, updating each component for each game object will have O(n*m) time complexity (n: number of game objects, m: number of attached components).
Although it's time complexity is linear, it can grow quickly which makes updation slower. To counter this, we will impart behaviours via scripts, instead of components.
To make make runtime more efficient, PiEngine loads all the required resources when the engine initializes. PiEngine is using a precompiled header as well to make compilation faster. Special care is taken to include in the precompiled header; only most commonly used core stuff used by most of the systems in the Engine.

Singleton Managers:

Above the Component layer, our engine has Managers which are globally-accessible, non-copyable singletons. They have public getters for easy access to data that any system can read. The active Managers are:

- Input Manager
- Frame-rate Controller
- Resource Manager
- Memory Manager
- Event Manager
- Threads Manager
- Audio Manager
- Debug Manager
- Scene Manager
- Imgui Manager
- AI Manager
- Object Factory
- GameObject Manager

Resource:

When Pi Engine initializes, a config file is read which contains info about the initial states of the systems. It also contains asset directories, which is parsed recursively by our file parser. Resource Manager then refers the information obtained from file parser to load all the resources and maintains a lookup map for the game to use. The idea is to have no loads in game loop which can affect the frame time.

Rendering:

Pi Engine uses modern OpenGL for Hardware Accelerated Graphics. The Rendering engine currently supports 2D sprites i.e. it can render pngs, jpegs and bmps. Pi Engine also supports 3D Models and can load models of various file formats like FBX, Collada Files and OBJ files. We have implemented Texture Cache to avoid loading a single texture multiple times and if the same texture is loaded then a reference to already loaded texture in the GPU is returned.

Renderer creates render commands at initialization for primitives such as Triangles, Quads, Cubes and Spheres where VBO, VAO and EBO are bound. So only a single instance of one type of primitive is used to render multiple objects of that type. So, our Renderer provides a Mesh Renderer, which can be attached as a Mesh Component to get Models or Sprites Rendered.
The Engine currently supports both Phong and BRDF lighting model but for our game we are using BRDF. We support three types of Lights:

- Directional,
- Point Lights and
- Spot Lights.

We also have support for 3 variations for camera creation that are:

- Free movable camera,
- Main Camera and
- UI camera.

Debug:

We are using visual studio's own task list to keep track of our "Todo tasks".
To add a task we just add "//TODO: task_description"
If we have hacks, we add "//Hack: hack_description" above the hack (although these are removed immediately after having the hacks used for temporary purpose).

Physics:

Pi Engine supports a 3D Rigid Body Physics system that is overall based on the Bounding Sphere Tree model with Rigid bodies deriving movement from velocities and forces.
The Physics system supports Rigid Bodies, which are derived from the RigidBody component. RigiBodies have basic movement based properties like position, velocity, acceleration and the list of forces acting on the body. The RigiBody component also has the collider variable that tell what kind of bounding collider the RigidBody has. At the moment, the Physics Manager supports the following colliders:

- AABB,
- Boxes,
- Spheres and

- Planes

The engine supports the following forces that act on Rigid Bodies:

- Gravity,
- Friction,
- Damping and
- Torque

The Physics Manager also supports breaking of RigidBodies into smaller similar bodies. The new bodies are spawned on collision and gain force based on the position and direction of the force applied to the body.
We shall further be working on introducing additional forces into our engine based on what the game may require. And if time permits, and the game demands, we shall introduce OBB collisions.

Imgui:

Imgui is being used for displaying Debug information such as Frame data, Camera Position, Light Position and so on.
Imgui will also be used to spawn Lights at different positions, change colors of lights and to spawn game objects as per request.
Once our engine supports scripting, we shall use IMGUI to make run-time changes to the scene.

Audio:

Pi Engine makes use of the FMOD API for its Audio and supports the following features:

- The Audio Module for the Engine supports **3D Audio** with the following sub features:
    - **3D Listeners** with audio fade based on distance.
    - **Effect of Velocity** on the audio.
- Audio in the Engine is loaded into either a **Music-channel group** or an **SFX-channel grou**p making it easier to control their output.
- **Basic Mute, Pause** and **Stop functions** for channel groups as well as the overall system.
- **Slow-Motion Audio** effect to support our game's potential Slow-Motion mechanics.

Input:

Pi Engine's Input Manager makes use of the SDL2 API. We are using SDL2 version 2.0.9 in the engine and all input is drawn from SDL's Event system. Pi Engine has support for Mouse Input, Keyboard Input as well as Controller input.
All three devices support the three basic event states that are:

- Button Down
- Button Up and
- Button Held

Along with a few special trigger states that are as follows:

- Mouse:
    - Double-Clicked
    - Scroll-button scrolled up &
    - Scroll-button scrolled down
- Controller:
    - Left and Right Axis values in X and Y direction

- Left and Right Trigger values

The controller and the mouse have their buttons defined in enums and their values get updated based on SDL events. Based on how their previous and current states differ, we generate boolean events.

AI:

Pi Engine's AI system is State Machine base, that is its a system of various NPCs having multiple Behaviors, behaviors that set depending on what state the game object is in.
The Engine has an NPC component, components that are all managed by the AIManager. Each NPC has a behavior variable that assigns it any behavior that has been defined in the Behavior class. We use this behavior variable to give various game objects and NPCs their behaviors.

Event:

Pi Engine's Event manager currently holds basic event handling functionality:

- Engine Managers can create different types of events as needed
- Events can be instant or timed
- Event manager holds events queue and broadcasts accordingly to all game objects

Currently functional with the need of a minor testing and adjustments. An extra layer of Subscriber/Listener might be added eventually for better event communication between the engine systems if needed.

Memory:

Pi Engine's Memory Manager relies on object pooling, reserving required game objects and components only on initialize, then the objects are ready to be used and reused as needed by the game, to eliminate the constant use of new and delete. On exit freeing the memory pool is handled as well, still needs more testing and optimizing. Different objects will be added to memory pool as the engine progresses, and memory sizes will be adjusted accordingly to not over reserve.

Thread:

Pi Engine's Thread Manager will create a pool of threads on initialization. It also has an internal scheduler that will collect jobs from different systems in the engine, then distribute accordingly. Mutex locks are used for safe data access between the threads. The thread manager is functional but still requires a good amount of testing to guarantee correct behavior and distribution of jobs among threads, might need a few more design and testing iterations to reach a fully functional bug free state.

Object Factory:

Pi Engine's Object Factory is the production line that puts the parts together to create the game objects that will be used in current scene, It creates gameObject according to the archetype passed and adds components from the memory manager. Depending upon the values read from json file, it serializes the component with those values and return the pointer of constructed GameObject which in turn gets added in list of active game Objects maintained by gameObject Manager.Currently functional as needed, testing and adjustments might occur if needed.