Assignment 2: Single Player Engine and Game Design Doc

Group Members
Ka Seng Chou
Joe Vennix
Keegan Standifer

Status Update as of Demo Date (2-20-14)

For the past two weeks, we have been working on creating a Unity-esque game engine to build our games with. The structure of this system is that all elements inside the world are Game Objects. Each game object has a list of Components which add behaviors to a game object. Outside of this GameObject system, we have Managers that run the overhead and common duties of the engine, such as the physics simulation or sound.

The work that we have done so far has created an engine that is a completely separate entity from any game. In order to set up an application that uses our engine, you simply include TheEngine.h. An example of how this is done can be seen in our Application.cpp, which includes very few calls to set up the engine, and uses a very simplistic API to communicate with the workings of the engine.

Below is a report of all features and our current progress on them.

Managers:

- AudioManager - Written but contains major bugs.
- InputManager - Working and implemented into Demo.
- SceneManager - Working and implemented into Demo.
- PhysicsManager - Working.
- GraphicsManager - Working and implemented into Demo.
- GUIManager - Working and implemented into Demo.
- ResourcesManager - Working and implemented into Demo.

Components:

These all inherit from a component class.

- Camera
    - FPS Camera - Partially working. (Rotation but no movement)
- Rigidbody
    - SphereCollider - Written but not tested.
    - BoxCollider - NOT STARTED
    - Triggers - NOT STARTED

- Transform - Working and implemented into Demo.
- Mesh - Working and implemented into Demo.
- ParticleSystem - NOT STARTED
- LightSource - NOT STARTED
- AudioSource - NOT STARTED

Measures of Progress:

After the implementation of a standalone game engine, we started to construct the demo. We were able to get the game to enter the render loop that steps through rendering and physics. The game camera works and displays GUI that is created by the GUI manager. The Input manager was tied in so that the GUI can be enabled or disabled by pressing Q and W.

The game camera is placed using a transform, and the Ogre Head is shown using a Game Object with an attached Mesh Component.

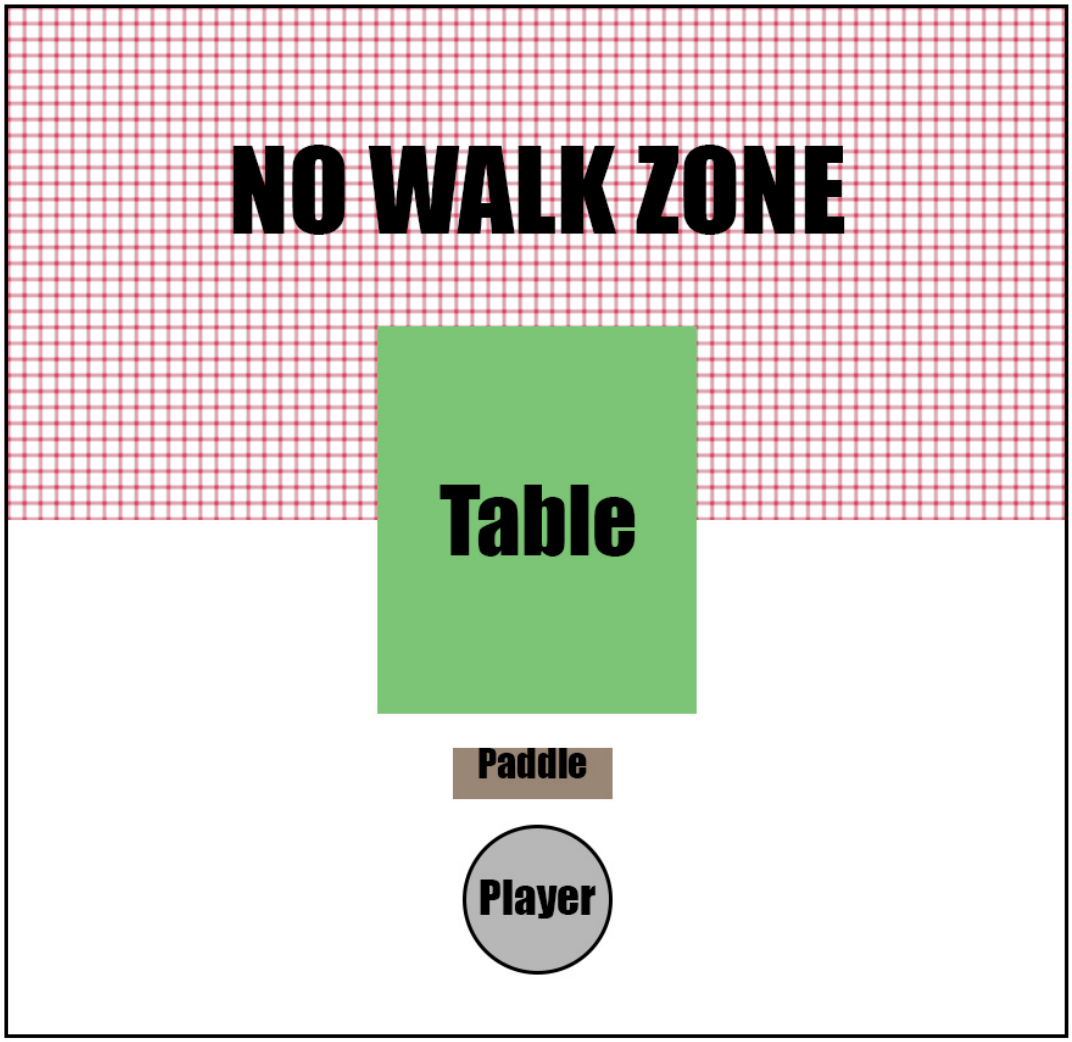
Now that the architecture for game components is implemented, and the graphics and physics managers are just a few tweaks away from being finalized, our simple and elegant API will allow us to finish the game at a greatly accelerated rate.

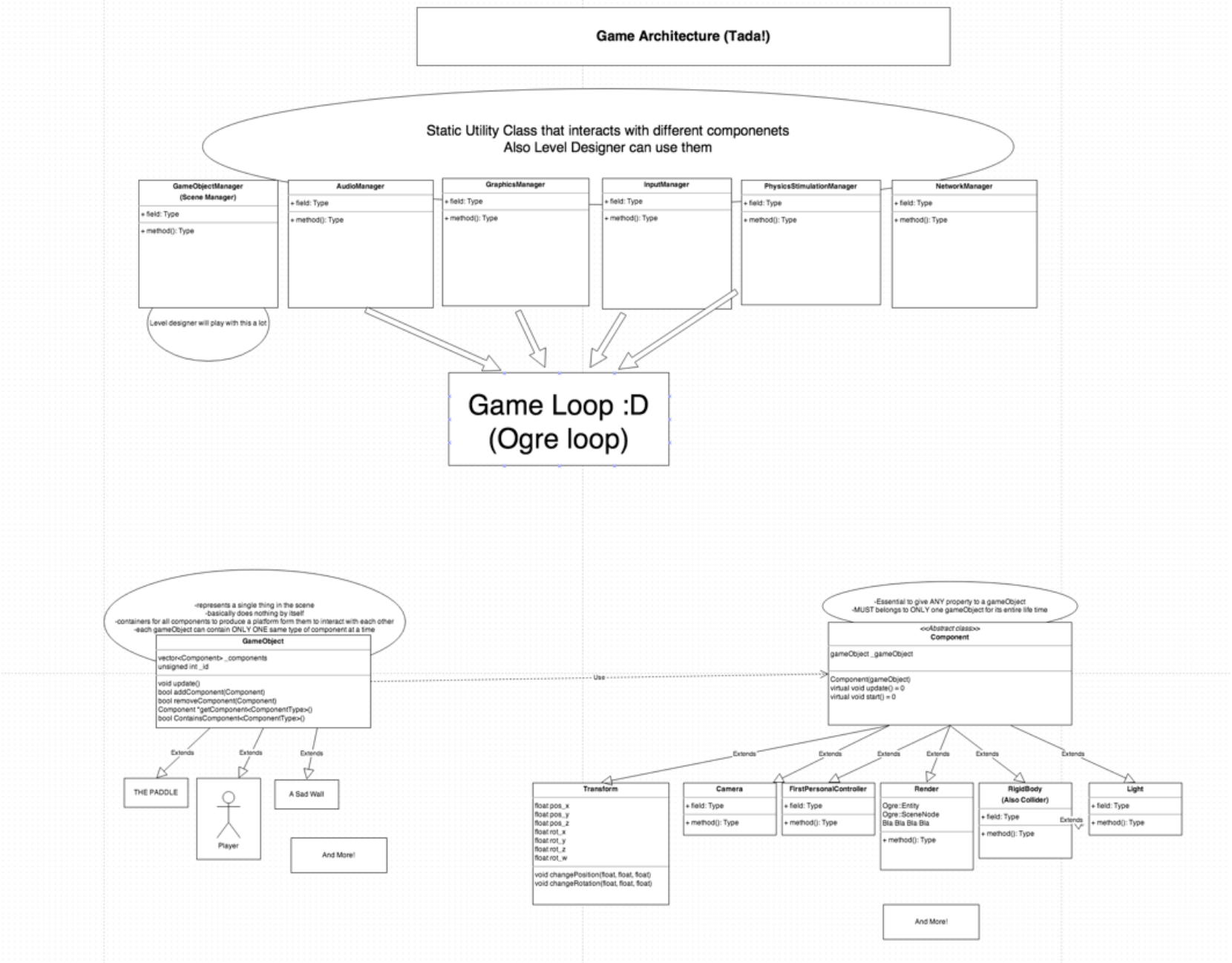
BELOW IS THE ORIGINAL DESIGN DOCUMENT.

For our game, we will implement a minimal 3D version of wall-ball, with a couple extra rules. The user will control a block-shaped “paddle” to bounce a ball off of a table and against a back wall. If the ball ever touches the ground, a point is deducted. Everytime you successfully bounce the ball off a wall and on to the table, a point is added. The controls will be WASD keys to move forward or turn, and the space-bar to “jump” the block forward. The game will take place in an enclosed room, and the user can bounce the ball off any wall or ceiling. The game physics will be similar to actual tennis (very elastic collisions), and the game will move at similar pace.

As far as software design, we plan on building a collection of static classes to manage “global” behavior - audio, input, the physics simulation, graphics, game state, and the scene. The individual pieces of the game (the block, ball, and walls) will be implemented as subclasses of a GameObject superclass. Shared behaviors between the subclasses will be implemented as “attachable” instances of a Component class.

In order to efficiently parallelize the work, we will be using git to work on separate components individually. We will be hosting the code as a private repo on github.com, which will let us review and stay up-to-date with each others’ code.


NO WALK ZONE
Table
Paddle
Player


Game Architecture (Tada!)
Static Utility Class that interacts with different componenets
Also Level Designer can use them
GameObjectManager
(Scene Manager)
+ field: Type
+ method(): Type
AudioManager
+ field: Type
+ method(): Type
GraphicsManager
+ field: Type
+ method(): Type
InputManager
+ field: Type
+ method(): Type
PhysicsStimulationManager
+ field: Type
+ method(): Type
NetworkManager
+ field: Type
+ method(): Type
Level designer will play with this a lot
Game Loop :D
(Ogre loop)
-represents a single thing in the scene
-basically does nothing by itself
-containers for all components to produce a platform form them to interact with each other
-each gameObject can contain ONLY ONE same type of component at a time
GameObject
vector<Component> _components
unsigned int _id
void update()
bool addComponent(Component)
bool removeComponent(Component)
Component *getComponent<ComponentType>()
bool ContainsComponent<ComponentType>()
Use
-Essential to give ANY property to a gameObject
-MUST belongs to ONLY one gameObject for its entire life time
<<Abstract class>>
Component
gameObject _gameObject
Component(gameObject)
virtual void update() = 0
virtual void start() = 0
Extends
THE PADDLE
Player
A Sad Wall
And More!
Transform
float pos_x
float pos_y
float pos_z
float rot_x
float rot_y
float rot_z
float rot_w
void changePosition(float, float, float)
void changeRotation(float, float, float)
Camera
+ field: Type
+ method(): Type
FirstPersonalController
+ field: Type
+ method(): Type
Render
Ogre::Entity
Ogre::SceneNode
Bla Bla Bla Bla
+ method(): Type
RigidBody
(Also Collider)
+ field: Type
+ method(): Type
Light
+ field: Type
+ method(): Type
And More!