Due: Monday, October 24, 2016, 8:00 AM (submit to BlackBoard, under Assignments)

File Type: Microsoft Word

Team Name:

Monster Master

Team Members and email addresses:

Cammy Vo - c715v349@ku.edu  
Kurt Slagle - kslagle@ku.edu  
Colton Roemer - colton.roemer@ku.edu  
Nicholas Roudebush - nroudebush@gmail.com  
Austin Bailey - a604b997@ku.edu  
Charles Thomas - Chuckt2@ku.edu

Team Meeting time:

Saturday 12:00 PM

Lab Meeting time:

Friday 4:00-5:50 PM

Contact:

Kurt Slagle - kslagle@ku.edu

Project Description (150-250 words)

- The project is being undertaken because we see an empty niche in the game market, and we want the experience and modularity that developing our own game engine provides. Most of Nintendo’s pokemon games are sold to adults who played them as children, and to create a creative, fun, more mature game for them that can be handled on a mobile device requires a dynamic, well-designed sleek engine.

- The end result of the project is a functioning engine with a basic game on top which can be easily scaled because of engine support

Project Milestones

**1st semester**

- Wiki update: Write pseudo about engine and game, along with its functions. Document Engine frameworks, both current and planned, and usage. Expected Completion: 10/8/2016
- Graphics Milestone 1: Use the engine combined with textures to generate a one-layer level consisting of a single sprite representing the character and a number of other
sprites representing tiles that can be crossed and cannot be crossed. Expected Completion: 10/20/2016

- **Graphics Milestone 2:** Animated sprites. A sprite sheet can be loaded into memory and used to animate a sprite on-screen, which may loop the animation if desired. Expected Completion: 10/24/2016

- **Graphics Milestone 3:** Three layer level with changing sprites for each direction including frame sliding for movement which uses the moving sprite for its duration. Expected Completion: 11/11/2016

- **Game Milestone 1:** Useful shop interface and implementation in a console-based version of the game. Expected Completion: 10/21/2016

- **Engine templates for Actors, Items, Weapons**

  **2nd Semester**

- **Port to android:** Have the game capable to be run on mobile devices and place the game on the app store. Expected completion: May 2017

- **Engine:** Particle effects can be used in-game Expected Jan 2016

- **Engine:** Physics system. The engine can simulate some physics, such as rolling and bouncing items. Expected Feb 2016

- **Engine:** Dynamic Lighting based on current level/terrain. Expected completion: March 2017

- **Engine:** Long event sequencing/Cutscenes. Expected Feb 2017
- **Gantt Chart**

**Project Budget**
- Hardware, software, and/or computing resources
  - Hardware: expected to port to android
  - Software: Uses SFML 2.4.0, can be coded on any C++ based compiler
  - Other computing resources: none
- Estimated cost
- 25$ (fee to put a game on android app store)
- Vendor
  - Android app store
- Special training (e.g., VR)
  - None
- When they will be required?
  - The vendor cost will be needed at the very end of the project if we decide to post the game on the app store

**Work Plan**

- Kurt - Engine lead
- Austin - Game design/lead
- Cammy - Game I/O portion and maybe some parts of the engine
- Nick - Likely AI or multiplayer networking or something with database design.
- Colton - Flexible, game potion seems most likely
- Charles - Flexible, Animations, Engine

**Github link**

https://github.com/JayhawkZombie/EECS581Project/

**Preliminary Project Design**

**Overview:**

The project is designed to be both a game and an engine. The engine should be designed to be interfaced with a game that we’re designing, as well as other games if possible. The game is a basic role-playing, single-player game that may be extended into a MMORPG.

**How the software works**

- **Engine**
  - The Engine side uses SFML for rendering, audio, and networking using 5 main modules: System, Window, Graphics, Audio, and Networking
    - The main Engine class contains the main logic to handle control flow, timing, window management, and thread execution.
    - The Engine is designed to be modular, so pieces can be added without immense complications in the main engine logic.
    - For the main startup loop, initialization is done, window handles are acquired, and things like the resource manager are created and started by the engine, before any of your derived class objects are ever created.
      - Some resource are made to do diagnostics and timing, but the main operation sequence is:
        - Poll Events (get events from the OS queue)
        - Updating timing (amount of time passed since the last frame was updated), and pass the amount of time passed since the last frame was updated, with double precision
        - Call TickUpdate for the main level, which in turn should call TickUpdate for all the objects in the level
o Call Render for the main level, which in turn should call Render for all the objects in the level
o Call Render on the UI overlay, if there is one active
o Repeat Loop
o Initiate shutdown
o Calls Engine::Shutdown()
  o Engine::Shutdown() calls OnShutDown() for the main level, which should do the same for every object in the level
  o The resource manager is then shut down, releasing all currently pooled resources and closing any open file handles
  The main thread waits for the resource manager threads to return (.join())
  o After all other threads have returned and all resources are released, the program returns. 0 should be returned if no error occurred during the shutdown process.

- In the Base Class, the highest-level that the engine can directly communicate with, 3 methods are contained in the class that must be overwritten by anything that derives from it.
  - void TickUpdate(double delta): Called every frame, with the amount of time passed since the last frame was updated
  - void Render(): Called every frame to tell elements to render themselves
  - void OnShutdown(): Called when the engine is shutting down to tell elements to clean up their resources
  - Every instance of an object derived from this contains an EventHandler object, which can be used to notify the object that an event occurred
  - Each base class method comes paired with a function pointer with the same signature, used for binding its event handler to those methods.

- For the Base UI, the elements contain a number of methods, as well as function pointers that can be bound for custom behavior.
  - The class derives from BaseEngineInterface, which means that every element contains an event handler.
  - The callbacks in that event handler are used for the elements to do element behavior (like determining if the element was clicked on, was moused over, etc), and not for custom behavior outside of the element.
  - The element contains a number of function pointers that can be bound, and those are used for custom behavior.
  - Function pointers that can be bound based on user input.

- The UI Handler class, which handles managing event passing for a set of UI elements, has a single pointer to an element that is "focused". That prevents all other elements from being notified of CLICK events, but does
not prevent them from being notified of MouseOver or MouseLeft events. When an element is given focus (usually when a mouse is clicked on them), then the OnFocusGained method is invoked, and when focus is lost (when clicking outside of the element) the OnFocusLost method is invoked. There is no way to forcefully take focus at the moment.

- Each Element contains a set of MouseTargets, RenderTargets, and KeyTargets
- The RenderTarget contains a pointer to a drawable element that will be drawn when the UI Handler containing the element is told to render itself.
- The MouseTarget contains a rectangular region in which the UI Handler will check for event occurrences. If an event occurred outside that box, the element will not be notified. Custom behavior can be bound to these.
- The KeyTarget signifies which key events the element wants to be notified of. If a key event occurs and the active (focused) element wishes to be notified of that event, the UI handler will inform it of the event. KeyTarget contains a bitset (bunch of bools) to tell the UI handler if this particular element cares about certain key events. By default, all key bools are FALSE.

- The EventHandler is the class in the Engine directly responsible for handling events.
  - The class contains a set of member methods, which the engine can call directly, and a set of function pointers, which can be rebound to invoke custom behavior.
    - There are 4 of these methods, each of which takes a function pointer of a different signature
    - These are called to bind the function pointers stored in the event handler
- The Engine contains a multi-threaded resource manager and loader. Loading is done in a thread that is not the main thread so that the Engine can continue to operate smoothly.
  - The Engine can only respond to events and render from inside the main thread, so blocking the main thread with file IO will stall the main thread and can lead to a multitude of negative issues:
    - FPS Stuttering
    - Slow animations
    - Missed events
    - Barrage of events that piled up
    - Process marked unresponsive
    - Process killed
- The resource manager separates all file IO in a different thread and has a dedicated thread that can be woken up when there are new resources in the queue.
Performance is critical. Due to the low-level nature of the Engine, some intensive initialization practices are avoided to save time and the initialization deferred until the resource is needed.

The engine is due to overhaul the default memory allocation scheme and implement its own global resource manager to speed up allocation/freeing of memory, since C++’s default methods can be slow for large amounts of allocation/freeing in a short period of time.

Rendering
- The engine is ultimately using OpenGL to do rendering. While SFML itself only requires OpenGL 1.1, the engine will use features up to OpenGL version 4.3.
- The rendering engine fully supports mixing of direct opengl calls and SFML abstracted rendering, this 3D and 2D can be rendered easily with little extra work.
- The rendering engine fully supports GLSL, so complex shaders can be written and used, for things like post-processing and dynamic lighting.

The Game side is coded in C++ along with the engine and the code is divided into Fight, Character, Item, Monster, and Travel classes.
- The assets of the game (Characters, Items, Maps etc) are soft-coded (read-from-file).
- The game has further subdivided classes into the types of characters and what their functions are, items and what those items will do, terrains and what those terrains will do, etc.
- Each new class has a separate function and sometimes contains dependencies and derivations. The classes are designed to be mostly modular.

Figure 1. Engine UML Diagram
Figure 3. Game UML State Diagram
Ethical and Intellectual Property Issues

- **Ethics:**
  - This project does not violate any utilitarian nor respect for persons tests. As far as utilitarian tests go, the project only aims to increase utility among the older population who has experienced playing games in the past and wish to relive old gaming memories through a new mature platform, as well as implement and design an engine that will benefit the engineers’ experience and other games that will want to use it since it should allow gaming developers a wider selection to choose what to implement their game on. As for following the ethics of respects-for-persons, although the game will borrow elements from pre-existing games and could possibly be seen as violating a second-tier right, to have possessions stolen, with a third-tier right, property, in some cases, the game does plan to be unique in its own right.

- **Intellectual Property:**
  - Since the project will be fully designed on our own using our own code and ideas, with only loose basing off of other games, there should be no violation of Intellectual Property. Thus far, the game and engine will operate under a general copyright policy under the University under self-initiated mediated courseware and does not plan to borrow from any other software, though it does use SFML. SFML operates under a Zlib open-source license which allows anyone to use or modify it under any kind of commercial or personal use, as long as they do not attempt to claim writing the original SFML code or claim a modified SFML library as being the original.

Change Log

- Hardware, software, and/or computing resources changed to be more precise on what was already expected.
- Work plan was changed because of differing amounts of familiarity with the coding
- Changed the milestones because expected ambitions and actual work on the project were different