Team Number:
   13

Team Members and email addresses:
   Kurt Slagle            kslagle@ku.edu
   Colton Roemer         colton.roemer@ku.edu
   Cammy Vo              c715v349@ku.edu
   Nicholas Roudebush    nroudebush@gmail.com
   Austin Bailey         a604b997@ku.edu

Team Meeting time:
   Saturday 12:00 PM

Lab Meeting time:
   Monday 2:30-2:45

Contact:
   Kurt Slagle - kslagle@ku.edu

Project Sponsor: None

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 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
● 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 March 2017
● Engine: Physics system. The engine can simulate some physics, such as rolling and bouncing items. Expected Feb 2017
● Engine: Dynamic Lighting based on current level/terrain. Expected completion: March 2017
● Engine: Long event sequencing/Cutscenes. Expected April 2017
● Game: Bestiary Console Implementation. Expected Early Feb 2017
● Game: Shop Console Implementation. Expected Feb 2017
● Game: Reliquary Console Implementation> Expect Late Feb 2017
● Game: Status Menu Console Implementation. Expected Late Feb 2017
● Game: Battle Console Implementation(simple). Expected Early March 2017
● Game: Witch Console Implementation. Expected March 2017
● Game/Engine: Working Save File system. Expected Late April 2017
● UI: Implement Conversation March 8
● UI: Implement Bestiary March 15
● UI: Implement Shop April 12
● UI: Implement Battle April 26
● UI: Implement Witch May 3

Project Budget
● Hardware, software, and/or computing resources
  o Hardware: expected to port to android
  o Software: Uses SFML 2.4.0, can be coded on any C++ based compiler
● Estimated cost
  o 25$ (fee to put a game on android app store)
● Vendor
  o Android app store
● Special training (e.g. VR)
  o None
● When they will be required?
  o 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 - Engine Developer
● Nick - Likely AI or multiplayer networking or something with database design.
● Colton - Flexible on game and/or engine side; Video Production/editing

Github link
https://github.com/JayhawkZombie/EECS581Project/
Final Project Description

Preliminary Project Design

The project is designed to be both a game and an engine. The engine should be designed to integrate game content that we're designing, as well as other games if possible. The game is a basic role-playing, single-player game.

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
    - Call Render for the main level, which in turn should call Render for all the objects in the level
    - Call Render on the UI overlay, if there is one active
    - Repeat Loop
    - Initiate shutdown
    - Calls Engine::Shutdown()
    - Engine::Shutdown() calls OnShutDown() for the main level, which should do the same for every object in the level
    - 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())
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. UI elements can forcefully take focus if necessary, such as showing an alert the user must handle.

- 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:
    o FPS Stuttering
    o Slow animations
    o Missed events
    o Barrage of events that piled up
    o Process marked unresponsive
    o 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.
  o Performance is critical, but so is safety. To avoid memory issues and potential destruction of elements too early, all dynamically allocated assets are reference counted using std::shared_ptr from the C++ STL.
  o The engine leverages portions of the C++ STL due to its flexibility and ability to efficiently move objects around.
  o The engine has incorporated a scripting engine, Chaiscript, which is used to script behavior. Portions of the engine’s API are exposed to the scripting engine. Everything that is done via script can be done in C++ Code.
• 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.
  o The assets of the game (Characters, Items, Maps etc) are soft-coded (read-from-file).
  o 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.
  o Each new class has a separate function and sometimes contains dependencies and derivations. The classes are designed to be mostly modular.
  o The Game classes take advantage of the inherency property of object-oriented languages which allows for organization of Game elements based on what class the element is derived from.
Figure 1. Engine UML Diagram

Updated engine UML diagram

Figure 2. Game Use Case Diagram

User Interface

Start game

<<include>>

Display graphics and music

Go through events

Move player

<<extend>>

Close game

User

User 2
Figure 3. Game UML State Diagram

Figures 4 - 6: Engine testing Screenshots

Lighting Sample

Early UI Testing
Physics testing (including updated UI)
Ethical Issues

- 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 engineer's 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 ideas from pre-existing games this is not considered.

Intellectual Property Issues

- 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, the written software will be the IP of the members writing it, as permitted by University of Kansas policy, 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.
- Some ideas used in the engine have been borrowed from Unreal Engine, which is fully permitted in the open-source license provided by the developers.
the engine integrates content drew inspiration from Unreal Engine. As quoted from the Unreal Engine FAQ:

Can I study and learn from the Unreal Engine code, and then utilize that knowledge in writing my own game or competing engine?
Yes, as long as you don’t copy any of the code. Code is copyrighted, but knowledge is free!
https://www.unrealengine.com/faq

- Idea Origination: The original idea that our game improves upon can be found at the following URL: http://www.fighunter.com/?page=game&g=BSGN. Some of the concepts have been duplicated, and most have been changed and altered and are now unrecognizable however the fighunter.com game was never finished, does not seem to have been worked on since 2009 and none of the assets (algorithms or content) were duplicated and attribution for the original idea will be provided for in the credits, in addition to a link to the owner’s site.

Change Log

- Charles taken off the Work Plan and other parts. He’s no longer part of the team since he didn’t enroll in EECS 582.
- Changed lab meeting time
- Changed Procedure for item content generation
  - Initially the items were going to be hand written, however that is not scalable. As such there is now a tiered item system that is procedurally generated for the following subclasses of items: Weapons, Armor
- Changed Procedure for level generation/Reading by engine
  - We used to parse large text files to generate levels, however that became prohibitively time consuming and tedious. Now we are making and utilizing a level editor to make level content generation and execution simpler and less wasteful.
- Rewrote UI Framework to make it more easily extended and integrated. Allows for simpler nesting and combining of elements.
- The method by which the engine integrates game content was changed. The engine now uses an editor to integrate new classes into its code base. Game designers make the engine aware of the class by adding it and can then use the class with other content that has been added.
- Cammy has been moved from Game development to Engine development.