Project Proposal

- **Project Description:**
  My project, named Pydoom, is meant to be a videogame in the style of 90's first-person shooters such as Doom, Quake, and Duke Nukem 3D. It will feature at least one interactive level as well as a variety of enemies and items in a semi-3D world with a graphical style reminiscent of the 90's classics

- **Competitive Analysis:**
  Because the inspiration for my project was classic 90's first-person shooters, I spend the majority of my time investigating Doom and other similar titles, as well as more recent game that have been made to capture the nostalgic feeling of these games. There seem to be a variety of concepts present in these games that set them apart from modern action titles, such as the lack of a complex storyline in favor of fast-paced action, as well as the heavily-pixelated graphics (that were a hardware limitation at the time, but now are just for effect). Also, games such as this included simple lighting effects which enhanced the quality of the game while still being able to run at a high frame rate on their target machines. This rapid action and simple-yet-entertaining visuals are what I am looking to replicate in Pydoom. Also, like the classics, I will be writing my own 3D engine for the game, as well as a 3D entity and world map storage system.

  There are several differences as well between Pydoom and the games that have inspired it. One of the most major is the nature of the graphics engine itself. While Doom and similar titles had to be made specifically to be run on the machines of the time, most of which entirely lacked an independent GPU and therefore had to rely on memory manipulation tricks to achieve a playable framerate, Pydoom has the advantage of being able to use a more modern rendering pipeline. Because a powerful CPU-rendered game would be incredibly challenging in a higher-level interpreted language such as Python, I have chosen to take advantage of modern tools such as Python's OpenGL bindings, PyOpenGL. This allows me to push all of the rendering calculations to the GPU, lifting the burden of actually drawing the scene to the screen from Python, however still giving me full control of my custom 3D engine. Also, despite taking heavy inspiration from Doom, it will have different levels, graphics, audio, and etc.

- **Structural Plan**
  The structure of the game will be broken up into five different Python files, as well as a 6th Python file specifically for converting world maps into a readable form. They are as follows:
  - **Main**
    Consists of all of the initialization that needs to be done, as well as the game loop itself
  - **Entity**
    Consists of an Entity class as well as a child class for every dynamic item in the game world, such as the player, enemies, items, etc.

- **World**
  Consists of a World class, which consists of the way to store the collision data for a particular world map, as well as all of the Entities that exist inside said world map
- **Mathematics**
  Consists of all of the classes necessary for doing 3D calculations and rendering, such as 3D and 2D vectors
- **Assets**
  Consists of all of the items that have to be loaded into the game during initialization so that they do not have to be continuously loaded every frame, for example textures, world maps, and 3d models
- **Converter**
  Is only used when an update has been done to one of the levels, it takes the data in from the level map (which is stored in slices of the level) and converts it into a file that can be read directly as an instance of a World map

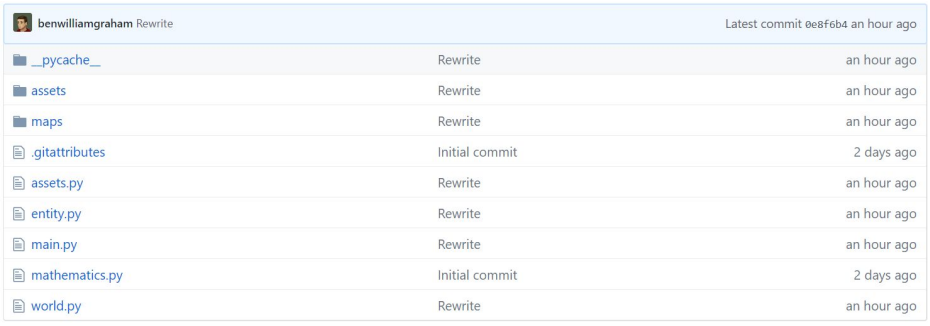
In addition to the Python files, Pydoom also has level files and textures

- **Algorithmic Plan**
  Although there are a variety of algorithmically complex operations that need to be completed, such as map meshing and static shading, pathfinding, collision detection, and rudimentary physics, the trickiest part of the project is the 3D engine itself. It will be written using PyOpenGL, the OpenGL bindings for Python. To render everything to the screen, several things will have to happen first. Beyond the obvious step of opening a window, it will set up a 3d perspective matrix, then allow the GPU to accept and transform 3D vertices, texture coordinates, and vertex colors. Then it'll enable and set the alpha function to exclude all pixels from the depth buffer which are transparent. Before the rendering can start, all 3D entities are bound to the GPU using a VBO (vertex buffer object), which includes the coordinates in 3D space, the position in the texture, and the color bias of each vertex (which is used to provide static shading). The access value is saved so that this will not need to occur every frame.

  In the game loop, the screen is first cleared. Then depth testing is enabled so that things that are closer to the player will be drawn on top of the things that are far away. Next, a transformation matrix is pushed to the GPU, which transforms all following entities into the player's coordinate system (with the player at the origin, x being horizontal, y being vertical, and z being the distance from the player in the direction the player is looking). Next, the VBO and texture of the world are bound and rendered to the screen with no further transformation. Following that, all dynamic Entities are rendered to the screen after pushing a model matrix, which transforms the location and rotation of the Entity based upon its internal position and rotation relative to the player. Finally, the GUI is drawn to the screen by popping the transformation matrix and disabling the depth testing so that they are drawn to the screen in 2D. Finally the screen is shown.

- **Timeline Plan**
  My timeline is as follows:
  - **TP1**
    - Working 3D engine
    - Collision detection
    - Enemies
    - Particles
    - Multiple connected rooms
    - Basic audio
  - **TP2**
    - Full game
    - Combat
    - Score keeping
    - End goal
    - Items
    - Basic title screen
    - Better audio
  - **Final**
    - Multiple levels
    - More enemy types
    - More item types
    - Better title screen

- **Version Control Plan**
  Pygame is backed up to a [github repository]



- **Module List**
  math for sin and cos
  pygame for screen, mouse, and key control
  random for randomizing enemy movement and particles
  OpenGL for rendering
  ctypes for use with VBOs in OpenGL
  os for window positioning

- **TP2 Update**
  There is only one difference between my past and current plan for TP2. Instead of implementing a score system now and updated sounds/music later, I updated the sounds and added music now and will implement a score system for the final submission. This choice was made because I felt that at TP1 the sounds were more important for an MVP than the supplementary gameplay elements and therefore updated them first to improve the overall game experience at TP2