To fully develop this game we split the project into two main pieces. The first piece is the framework for managing the game. This framework will be our game engine and it will be implemented as a C# library that another XNA game project can attach. Ideally the framework will be written as generic as possible so to accommodate any game, including 2D and 3D components. The engine needs to be able to handle multiple game states and multiple in-game windows per state. Game states in our definition will include a particular state of the implemented game. Examples of game states would be: title, menu, game. The engine should allow transitions between two states, allowing us to add fade effects or other transitional effects. A state is more or less a place to put a bunch of state specific initialization or shutdown code. Each state has an update() and draw() that the engine calls at the appropriate times. Now even though a state can rendered on its own and a game could be implemented within a state the idea is to have states define windows. These are not traditional windows you see on your desktop, instead they are in-game windows. These windows are used to render sections of the game user interface in a controlled manner. It is doubtful our game will make heavy use of more than one or two windows but some examples might be: the game window, in-game pause window to simply fade game, in-game settings, perhaps in a mmorpg the inventory/bag space would be a window. The framework provides abstract “Basic” version of State and Window for users of the framework to inherit from and define the abstract method. Windows will contain components and components are objects that can be updated and/or rendered by the game engine. The framework provides interfaces and abstract classes for Component classes. Another important part about the framework is user input. Since user-input is best handled in a last state compared to current state method this is all abstracted for the user of the engine. The idea is each input device has events for pressing a button, releasing a button, holding a button. The input device will determine which events to fire whenever it is asked to process its input. It seems best to have states manage the input classes but technically they could be anywhere. Best practice though is to have a state contain the inputs you need, bind the events and have the state determine when to update the input state and process it. As for input events, these events will be keyed by the button pressed and the owner of the event. This way a state can bind events from multiple windows to a multiple buttons. Input events can return true to cancel the bubbling of the events or false to allow it to bubble. Examples of usage here might be having a particular in-game window open that handled mouse clicks like the inventory, clicking in this window, the state would do the collision detection, process the input for the inventory window which would prevent the mouse click from going through to the actual game window behind the inventory. An important part about every engine is how it handles Physics, collisions. To provide a decent framework that is not to restricting the following will be done. A unit is something that inherits from components and is added to windows. A unit provides a Physic packet from the valid state to the new state. All updates on a unit will save the valid state and attempt to update the new state. Each unit then is compared to a list of other units provided by the game, not the engine. The game implementation has to determine which units to test for collision against other units. Units are then responsible for returning a list of points to test for collision on. These points can be vertices or other collision objects like bounding boxes or spheres. The engine will provide all the basic collision methods we will need. However, when the engine finds a collision it will call a function provided by the unit class that handles the next step of collision un-related to physics. This would be things like activating a power-up, touching the final goal or other events that occur after collisions have been dealt with. The final core element of the framework is handling resources like imported graphics, shaders, fonts, sound and music. The engine will provide an easy way to get at XNA’s Content class that is used to load pretty much anything you can name. To facilitate shutdowns or reloading of resources states, windows all define destroy() and init() used to guarantee they have and release the resources they need. When resources are loaded into the engine they will be keyed with a simple string name. This will create a frustrating list of magic numbers, but the game implementing the engine can define an enum to cover all the textures, sound, etc they might want to add. This means we do not have to pass instances of loaded resources around to all the classes that might use it, instead a class can ask the engine for an instance of a resource by name.

* Please talk about the level editor.