# Game Overview:

The team’s proposed game is the creation of a “museum” like environment where they can showcase different elements/requirements of the assessment in the environment. This can be done in either separate rooms or a single larger room. The environment will be used to demonstrate the following:

* Camera:
  + The game will be played from a first person perspective, similar to a First Person Shooter.
  + The camera is attached to the player head(or where the head would be)
* AI:
  + “Visitors”, who are wandering characters with random/specific paths and will try to avoid blockers on their path
  + A following entity, possibly for the resemblance of “a dog” that follows the player
  + An entity that on player collision will promptly apologise and go on a new walking path
* Exhibits:
  + Will be used to demonstrate different areas of study: lighting, shadows, collisions(also available in other parts of the game) and/or different kinds of shaders
* Collisions:
  + Collisions will be available through the environment
  + These will be used sparingly, avoiding their use in areas where players shouldn’t be able to reach
* Interaction:
  + The player will be able to interact with various exhibits in the museum
* Graphics:
  + There will be entities with different models in the environment (exhibits, objects, entities, etc.)
  + Different types of shaders will be used, from more simpler ones such as Pixel/Vertex shaders to possibly more advanced ones made by the team
  + The environment will feature lighting and shadows

This leaves the project open to possible additions if the time allows it. These can be:

* Sound, both for the environment and its entities
* Procedural generation of rooms
* Animations

# Game Breakdown:



The game was split into reasonable sized chunks and elements deriving from each other. Considering previous experience (Computer Games Assignment – Develop a 2D Game) the team has decided that the graphical side (which is also a more DirectX intensive side) will require more work and time to proper develop.

Considering the above, the “Logic” side of the game, while covering more aspects of a functional game, may require less or equal amounts of time and effort comparatively to the “Graphics” side.

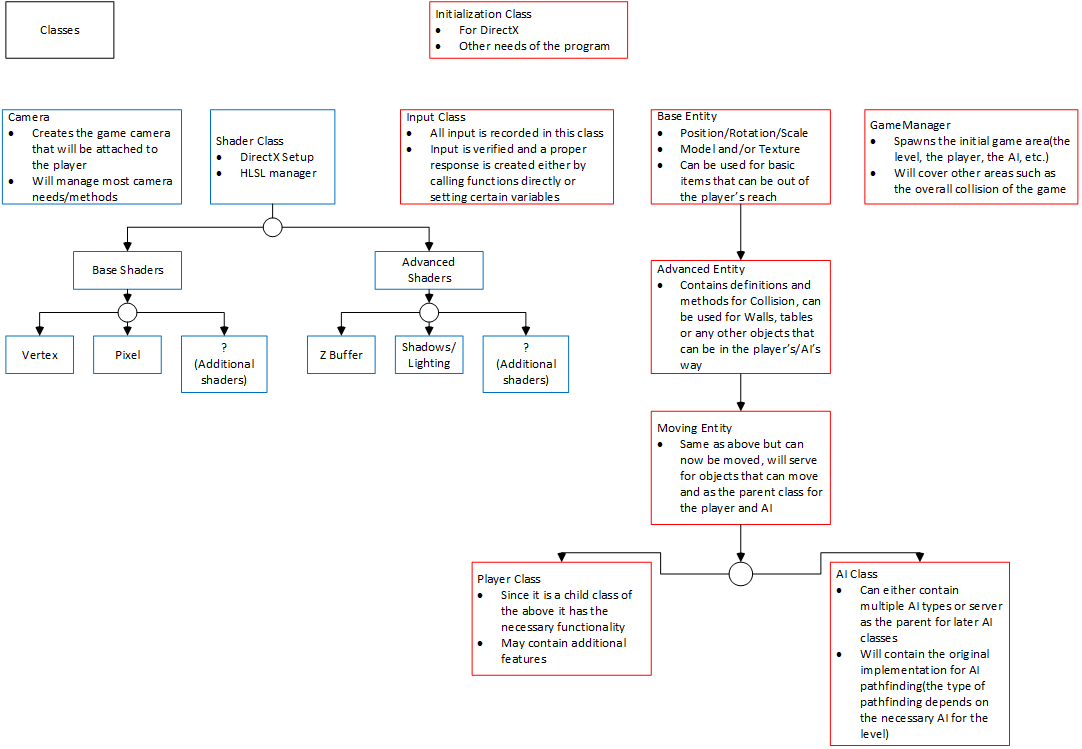
A single programmer cannot cover multiple areas of study, especially on big projects. We already see in the industry that there are different areas of specialisation: gameplay programmers, AI programmers, graphics engine programmer, sound programmer, etc.

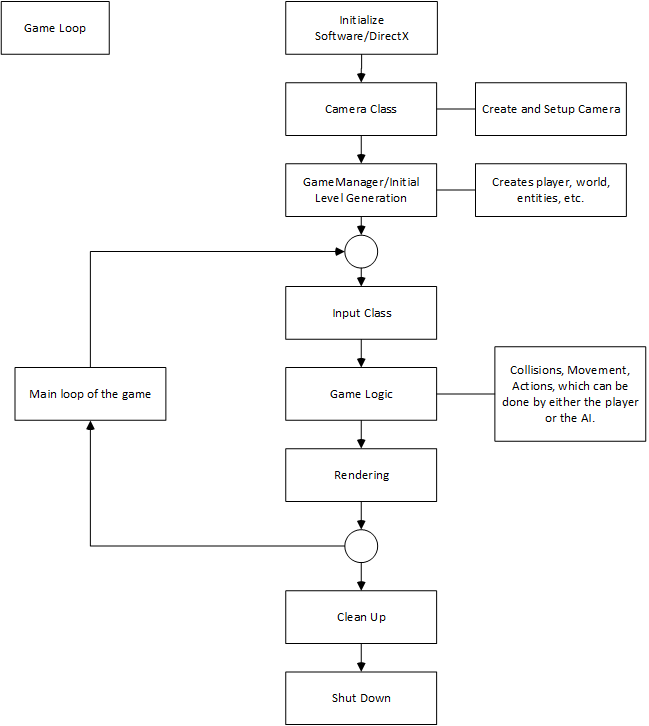
Using the above information as a basis the team has agreed to split the work as follows:

* George Alexandru Ciobanita will take the “Logic” part described above
* Nedelin Gochev will take the “Graphics” part described above

This way, members can focus on specific areas of study and research, without fear of confusion from covering too many areas. Thus members can provide proper information to each other and communicate new findings in a better manner.+

# Basic Class Diagrams & Game Flow:





There will be final updated version of the above as the team approaches the hand in date. These version will contain updated information from the resources gathered.

# Version Control & Log Keeping/Testing:

The following are proposed for better and safer work conduct:

1. Github, in order to keep files, versions and backup available online.
   1. Team members will upload files and information, with appropriate descriptions and titles, as they acquire it and keep it available in the “AGP-Assignment” repository.
   2. Members can create/update files as they see fit (there is a document available in the repository that each member has to update).
   3. Members will communicate with each other in order to clearly update any current piece of information. There is the possibility that the findings of one member can help the other realise mistakes or improve.
2. Visual Studio can be setup to create logs after the codebase has been compiled and tested.
3. For future use the team can employ the use of “assert” a function mainly used for testing. This can work well with the Visual Studio log implementation as the following are logged: the function expression, name of source file, and the line number where it happened (e.g. Assertion failed: *expression*, file *filename*, *line number*).
4. Grey-box testing, a combination between White & Black box testing, can be used through implementation.
5. The tutorials available from Advanced Games Programming give a good view in how to setup a project in order to properly get error/warning messages when certain systems do not function. This can be specifically seen in Tutorial 01 and 02.

# Initialisation Class

Other conclussions can be drawn from the same tutorials mentioned earlier. These focus on Windows initialization, Direct3D initialization and error handling for Direct3D.

From the tutorials we can note the following:

1. The Windows library is needed in order to created a proper Win32 window. From here we have different methods available in the library in order to affect different parameters(available in the class) that would change how the window will function.
2. Win32 applications, specifically videogames, work differently than normal Win32 applications, meaning that whether or not some event or action takes place, it will not keep the application from perfoming many tasks throughout its lifetime.
   1. This shows that the application will have a loop that runs continously until broken by the user’s input(usually by pressing the exit button).
   2. The loop will: record and dispatch messages, update the logic of the game, and draw the graphics.
3. The Direct3D section of the class and its methods need to cover the following: definition of device type and feature level(as no platform or software will use the same version of Direct3D), creation of the Direct3D device, rendering context and swap chain which can be specific to the project, creation of the render target and setting the viewport.
   1. For the purposes of the assignment and in order to give an easier time to the programmer, the setup available in the tutorials can be used as a starting point, which can later be updated as needs arise.
4. Obviously such things need to be followed by a clean-up method. Before closing the applicaion, resources need to be released(the Direct3D device, the Direct3D render context, the swap chain and the render target view are example only from this class, meaning there can be more added to this list).
   1. Objects need to be Released(), preferably in the reverse order of their creation and it is good conduct to check if objects are NULL prior to this method call.
5. Future proofing this class means setting it up to handle different errors. The DirectX library comes with functions and macros that can aid in debugging DirectX applications.
   1. As an example, DXTRACE\_MSG can be used in the context of checking whether or not the window has been initialized properly, this can reduce the time spent fixing errors.

|  |  |
| --- | --- |
| Initialisation Class | Conclussions: |
| * The first class of the program * Will have different method for initialisation, its loop and clean up * The class will server as the main game manager, as it is already created around managing events * Will also server as the final clean-up of the game once the application has been prompted to shut down | * Gamemanager and Input class may be dropped in order to better develop this class * The Initialisation class may need access to an array of variables(from other contexts such as AI or graphics) in order to better call or launch different methods * The time needed to develop and maintain this class may be smaller compared to other ones, as the topic of initialising directX and Windows applications is one of the more better covered areas, with resources available both online and physical documents. |

# GameManager & Input Classes

1. The Input Class

Since the application will be available on any other platforms expect PC, the conclussion is that input can be taken from keyboard and/or mouse. Since keypresses are handled as events by the system from the get go, one could make a case that this can be integrated in the Initialization class and its message queue system. However, reading input in such a way is slow and can affect the game’s performance in the long run, also it does not fall in the scope of the Initialisation class.

Including the above there are three ways of acquiring input information:

1. The message queue system integration.
2. Obtaining keystates via Win32 functions.
3. DirectInput.

Considering the project has the students develop in DirectX it makes sense to use the functionality available. DirectInput can be created similarly to any other DirectX object, and has to be released in a similar fashion, while requiring an input device. Using DirectInput can give the team access directly to the input device and will help with making life easier later on, as it makes it possible to read input from it each frame

|  |  |
| --- | --- |
| Input class | Conclussions: |
| * Will serve as a means of communication between the user’s input and the rest of the software * Makes sense to be its own class due to the fact that it is different in scope and developed in a different manner which would class with other classes * The input acquired will in turn call and affect the necessary methods of the Entity that the user plays as. | * Needs access to the Entity class that doubles as the user’s in-game avatar * DirectX SDK contains the Xinput API which supports a wide range of Xbox game controllers. This means that the possibility to add Xbox 360 Controller input to the game is open for extra marks. * When clearing and releasing DirectInput one needs to be careful enough to also release the input device. DirectInput can make it so input from the device only works in the actual game, and forgetting to do so can lead to devices being locked by the system, use Unacquire(). |

1. GameManager Class

Most, if not all, games contain states and state machines, which can double as AI. The way these are implemented have to be appropiate to the game. There are multiple ways of implementing such a feature:

1. Use different game states for different aspects of the game, with all the shared data in one place, perhaps a singleton that holds everything needed and is torn down when transitioning between states(e.g. from menu to play).
2. Another method is improving the above by adding sub-states. There are top level states, “Menu”, “Playing”, etc., which have their own state machines like “LoadingLevel” or “InGame”. Data can be specific to the “Playing” state object while sub-states control the flow of the game. This method can be implemented implementing individual and generic state machine objects or state stacking(linked lists).

It better to approach development of this class via the secondary proposal as it can show improvement of skills, as chances are both team members are already familiar with the first one. One can attempt at actually making a plan for the different states of the game(e.g. menu and its sub-states) but these can change drastically(e.g. members just don’t make a menu).

|  |  |
| --- | --- |
| Gamemanager Class | Conclussion |
| * Will most likely function throught the entire duration of the game(from menu to exit) * Depending on how the team implements the game it may need data from multiple sources(player, AI, level building and monitoring, etc.) | * The secondary approach may be more appropriate to implement and can promote a better code as sub-states can simply be added to sources that already have data |

# Entity Classes

This section will cover the basic Entity Class, the initial parent, and all of its children and will explain why such an approach was chosen.

The team has decided that not all objects from the game scene will require use of advanced implementations, this can be explained better with the following examples:

1. Base entity. Will be used mainly for objects that are out of the player’s reach. Contains implementation for models and textures, possible implementation to allow simple shapes instead(cubes, pyramids, etc.).
2. Advanced entity. Objects that the player can interact/collide with, will most likely be the most common items in a scene. May contain additional variables which will represent width and height in order to determine creation of a collision box.
3. Moving entity. Same as above but it can be allowed to move in a scene. In here main movement methods will be created which will be later used mainly by AI and Player classes. May contain additional variables in order to calculate movement properly(velocity, gravity, bounciness, etc.).
4. AI entity. Containing all of the above, this class would require methods which would allow it to detect objects in front of it and determine if collision will happen, thus altering its path with a different movement.
5. Player entity. Child of Moving entity. The actions recorded in the earlier explained Input class will corespond to the proper methods. The Player is a separate class as it may need additional implementation(e.g. scoring, health, interaction and possibly more).

It makes sense from an Object Oriented perspective to create these entities in such a way.

|  |  |
| --- | --- |
| Entity Classes: | Conclussion: |
| * The most often meet class in game. * Starting from the top each class will be developed more and more in order to better facilitate later classes. * This approach is better suited for this kind of game and may not be fit for other games. * Will allow for models to be added to each element or for simple definitions for simple shapes which can be created early in the life of the project. * Everything is based around 4x4 matrices. Position, scale, rotation, translation will be the main factors of a properly implemented game. This can be part of the main entity class, but if time allows it a maths library can be created for simpler use(and for future use in other projects). | * Initially entities will use colliders in form of bounding boxes, possibly bounding spheres. This may leave the project open for another additional features for more properly defined collision boxes. * While there will be gravity, velocity and other physics values and elements in the implementation of the game, these are done so the implementation is mathematically correct, as there may never be a case where the player will jump and gravity needs to affect the upward and downward directions of the player. * While each entity will be allowed to have a variable that shows what model/texture has been assigned to it, the implementation of actually showing this on screen will not be done in here as it is more “Graphical”. This means that entities will be part of the general array called for the drawing update section in game. |