**Table of Contents**

**1)** [**Getting Started in Polyngine**](#GettingStarted)

**I.** [**Preface**](#Preface)

 **i. Using Spaces and White Space**

 **ii. Where to Place Content**

 **iii. Do not use “/PolynGameEngine/”**

 **iv. The User Interface**

**II.** [**Directories**](#Directories)

1. **/PolynGameEngine/**
2. **/PolynGame/**
3. **Using Subdirectories**

**III.** [**Output Log**](#OutputLog)

1. **What is It?**
2. **System Log**
3. **Success Log**
4. **Warning Log**
5. **Failure Log**

 **2)** [**Basic Engine Pipeline**](#BasicEnginePipeline)

**I.** [**Class Hierarchy**](#ClassHierarchy)

1. **Creating Custom Classes**
2. **PController [Base]**
3. **PObject [Base]**
4. **PObject::PCamera**
5. **PObject::PStaticMesh [Base]**
6. **PStaticMesh::PSkeletalMesh**
7. **PStaticMesh::PCharacter**
8. **PObject::PLight [Base]**
9. **Plight::PPointLight**
10. **Plight::PDirectionalLight**

**II.** [**Renderer**](#Renderer)

1. **Renderer vs Environment**

**III.** [**Environment**](#Environment)

1. **What is an Environment?**
2. **Environment/Renderer States**
3. **Saving and Loading**
4. **Saving Breakdown**
5. **Loading Breakdown**
6. **Environment Pipeline**
7. **Creating Objects**
8. **Using Controllers**

**IV.** [**User Input**](#UserInput)

1. **How is Input Used?**
2. **PInputManager [Class]**
3. **PController [Class]**

**IV.** [**Materials**](#Materials)

1. **What is a Material?**
2. **Manipulating Shaders**

**1)** **Getting Started in Polyngine**

**I.** **Preface**

**i. Using Spaces and White Space** ALWAYS use underscores “\_” instead of spaces or whitespace. Using spaces in filenames can cause issues when saving and loading levels, objects, and items. In general, do not ever use spaces in strings that will be used when looking up or altering file paths (such as Texture and Model file paths). When making objects in the engine, most all places that allow space input translate the spaces into underscores “\_”, but do not rely on this.

**ii. Where to Place Content** When adding assets to your game, be sure to keep anything that applies only to the current project in the “Polyn Game” folder. Only Engine specific files, such as basic objects built into the engine, configuration files, Renderer files, etc… belong in the “Polyn Game Engine” folder. Please keep any game files out of the “Polyn Game Engine” folder. Making subdirectories within the “Polyn Game” folder is fine. Read more about content directories in the [**Directories**](#Directories) section.

**iii. Do not use “/PolynGameEngine/”** The “PolynGameEngine” directory is reserved only for engine files, not game files. The only reason you should use any file from this directory is to add, remove, or alter tools of the editor, NOT a project, game, or level.

iv. **The User Interface** The GUI for the editor has lots of features in different areas, all sectioned off so you can find them whenever you need.

* *Window Menu* The Window Menu lies at the very top of the window with File, Edit, Project, etc. This menu allows you to Save and Open Levels, Edit Project and Engine settings, exit the engine, and more. One important button here is the “Create” button. Clicking this button will present you with a list of all objects you can quick-spawn into the world.
* *Environment Toolbar* The Environment Toolbar is the rounded/pink bar at the top of the render window and just below the Window Menu. This toolbar is for changing how the render displays the final frames to you. Controls for this area include those such as rendering debug objects and lines, Fullscreen settings, and more.
* *World Hierarchy* The World Hierarchy displays all objects currently in the game world. Each of these objects has a button you can click on to select and deselect it. When selected, the object will populate its information in the Object Inspector (see more below for Object Inspector) and allow you to manipulate it in-editor.
* *World Editor* The World Editor is the window just below the World Hierarchy that displays a tabbed view of available settings to change world interaction.
* *Object Inspector* The Object Inspector displays all available information for the currently selected object in the World Hierarchy. From here, you have many buttons, sliders, and inputs to alter the world settings of the object. This is your window into the status of your levels, objects, and project.
* *World Settings* The World Settings contains all settings available for the world itself. The world excludes objects and implies the actual game-world that objects lie within. Settings for the world are things such as Ambient Lighting that will affect all objects within the world.
* *Camera Settings* The Camera Settings tab in the World Editor lets you change the movement and viewing settings of the render camera you use to see the 3D world.

**II.** **Directories**

1. **/PolynGameEngine/** directory includes engine-specific files. The main reasons to mess with these files are either to alter graphics functionality or alter low-level engine functionality and extend usability. This portion of the codebase should only be changed if you know what you are changing, and where all else in the code it will affect, otherwise it could be catastrophic. This directory may include (but is not limited to):
* Renderer
* Environment
* Base Classes
* Base Data Types
* Shader Files
* Engine Tools
* Engine Features
1. **/PolynGame/** directory includes game specific files; these files can theoretically be replaced with another project's files and function at full capacity. Changing the files here may potentially break game code if you let it but should not cause any major engine-graphics issues. Much of the core engine code and graphics codebase are hidden under two layers of engine API. This directory may include (but is not limited to):
* Custom Classes
* Derivative Classes
* 3D Models
* Texture Files
* Level Save Data

This directory houses content with some specifics to keep in mind. All game assets should be kept in the “/PolynGame/Assets” folder. Within this folder you will find some default folders that you can use for organizing files. These folders are not required to be used. You can make as many folders and subfolders as you want and place files in them. Most objects will display the path beyond “/PolynGame/Assets” for any item with a file path so you know where the file is that the object is using. All files mentioned above should always be kept somewhere in the Assets folder. Any files outside of Assets risk being left out when building the game.

1. **Using Subdirectories** You are free to make subdirectories within the “PolynGame” folder. Whenever you load in a mesh or save a mesh, the folders between “…/Assets/” and the Model/Texture will be saved to the level. You should see the subdirectory display change within the Object Inspector when you choose a new file for any part of an object and get updated to the new path. Do not place content inside of the “PolynGameEngine” folder, and surely do not make subdirectories in it. Many of the locations in the “PolynGameEngine” folder have automatic output directories and designated file paths to certain tools.

**III.** **Output Log**

1. **What is It?** The Output Log holds lots of nifty information that gives you a peak behind the scenes of the engine and renderer. Most actions any piece of the gameplay pipeline makes will give some sort of notice that it is happening. Once that action completes, you will often get a green "SUCCESS" message, or a red "FAILURE" message printed. Sometimes you will see a yellow "WARNING" label print; these labels are because something happened that did not cause any sort of failure but may not have been intended by the developer. This can often occur when the engine automatically completes an action for you to prevent issues. Many times, a "WARNING" message can be ignored, so long as the game functions as you intended.
2. **System Log** A system message is usually the engine telling you it automatically launched a process in the background without requiring you to approve, commit, or take action to enact it. This could be something such as the system showing a Log saying it is creating a file you tried to open because it did not exist. System messages are not warnings, only notifications, so you do not need to worry if a system message shows.
3. **Success Log** An output log prints a success message if a process that was launched completes with success. If there is no success message, there should often be a Warning or Failure message telling you what went wrong (see Warning and Failure sections for more information). Success means you have completed an action and it was able to finish all tasks required by said action.
4. **Warning Log** A warning in the Output Log is not necessarily a “FAILURE” but may potentially lead to one. The only time it is okay (not necessarily safe) to assume that you can ignore a warning is if you were expecting to see the warning. Often, warnings will show because of the editor being required to alter a task it was doing to protect the environment from failure. This action taken could be anything from skipping some execution code to a value not being provided, resulting in a field being empty.
5. **Failure Log** A failure message requires immediate attention and often means something went very wrong behind the scenes. These messages must be fixed, or they are likely to cause crashes, issues, or bugs later on. Failure messages often tell you a bit of information about why, how, or where the error happened, and the verbiage of Failures usually imply the area they crashed in (as each Failure is only used in one place in the code).

**2)** **Basic Engine Pipeline**

**I.** **Class Hierarchy**

PInputManager -> PController | PObject -> PCamera -> PStaticMesh -> PSkeletalMesh -> PCharacter | PObject -> PLight -> PPointLight, PDirectionalLight

*Any items below that are marked with “****[Base]****” are classes that are not intended to be created or used but have children that inherit from it that are intended to be created and used.*

1. **Creating Custom Classes** Custom Classes are a core component of the Polyn Engine. A class can be built based on any existing items in the engine, and the creation process has been built right into the engine itself to ease your workload. To create a custom class, you will need to navigate to the Menu Bar option for “Create”. Next you go to “Create -> Custom Class”. When you hover over “Custom Class” you will see a list of options appear; this list holds most all the classes you can derive your custom one from! To derive a custom class from a custom class, you may need to manually create the derived child. When creating a new base class such as a player, enemy, object, or scene interactable object, always use the Create menu and never hard-create your own in the code editor. When you use this creation menu, it automatically links and includes the necessary files for you and auto fills the functions that are required by the parent class type. When you pick something from the dropdown, you will see a save file dialog. Pick the directory to save the file to, just make absolutely, positively sure that you save all custom classes and Source Files in the “PolynGame” folder and no other base directory. You are more than free to save with subdirectories you create within the “PolynGame” folder, just don’t save the files outside of the project, like in say your “My Documents” folder. Any code placed outside of the project directory are not guaranteed to be included in the final build. When you create a new class from within the editor, it should automatically open your new “.h” and “.cpp” files in Visual Studio (or whatever software you set as default to edit C++ header and source files). Please keep in mind that the class will be named whatever you name your new CPP files (ex. “MyNewClass.cpp” & “MyNewClass.h” will make the class appear as “class MyNewClass” in the “.h” header file; this can be changed but is counter-advised because you never know what issues arise as a result of untested scenarios. Long story short, use the File menu “Create -> Custom Class” option to create a new file or class and it will automatically fill out all required code and functions, include all required documents, and setup functionality for you.

You can always include or access the newly created files and classes by referencing them as a normal C++ class once they are created. This is one reason why you may want to remember how to find the files you are creating in the Save File dialog.

1. **PController [Base]** This is a device that is responsible for interacting with the PInputManager to deliver keypresses to specific objects. Once you Possess an object with a Controller, that object can receive input commands from said controller. To interact with the controller, simply call to the PObject’s assigned controller and check for input; always error check to ensure that the Controller is valid before using it.

The Controller class communicates with the Input Manager to receive input. The reason it was abstracted is to give an encapsulation unit to the Input to assign player indices and implement future renderer features. Where the Input Manager simply gets all input and sets flags, the Controller only reads that input and tests it. This keeps input testing lightweight and easy to use. The controller also filters editor and game input.

1. **PObject [Base]** This is the most basic object allowed in the engine. An object simply provides a platform for implementation of 3D operations. Every PObject is equipped with World, View, and Projection matrices, and a full set of functions to manipulate them. PObjects are not intended to be created and used by developers in the engine, instead they are used as a base that all other objects inherit from. This inheritance is what gives all other PObject derived types their ability to move, rotate, scale, control, and be interacted with. This functionality should be expanded upon in custom classes.

A PObject may be spawned into the world and used for things such as attaching, moving, or any other custom feature. When I say that it was not a class intended for developer use it just means that it has no visible object or real interaction beyond basic functionality. Other classes are preferred unless it just bloats the game.

1. **PObject::PCamera** Derives from PObject. A Camera is an object, but one that can see the game world. The camera has everything a normal PObject has, plus a system for camera movement and interaction. Without a camera, the renderer does not know what to draw from. A camera acts as the eyes for whatever is being rendered. You may have multiple cameras and switch between which is actively rendering to create gameplay mechanics.

The render viewport is rendered using a camera that is only accessible while editing your project. The render camera is titled “RenderCamera” in the engine and is hidden from the developers and players. The only way to interact with this camera is through the Editor GUI.

1. **PObject::PStaticMesh [Base]** Derives from PObject. Static Meshes are 3D objects in the game world, with vertex, index, and texture data, that can not be animated or manipulated much. Static Meshes allow you to add 3D models to the game and render it in the renderer. No bones or skeleton. Only “.obj” model types are supported for Static Meshes at the moment, and animated meshes should be imported via “.fbx” file type. This is the base type for any PCharacter because collision requires mesh data. All PStaticMesh and derived objects have to ability to be collided with, but not to collide with.
2. **PStaticMesh::PSkeletalMesh** Derives from PStaticMesh. Skeletal Meshes are 3D objects in the game world and share all data with Static Meshes. The difference between Skeletal and Static is that Skeletal hold much more information such as animation data, bone information, joint information, multiple textures, and support for reading color data from vertices of the imported model. The supported model type for Skeletal Meshes is a “.fbx”. Skeletal Meshes have the ability to animate and move freely, while Static Meshes do not.
3. **PStaticMesh::PCharacter** A PCharacter is an object that derives from PStaticMesh to ensure vertex/index data is present. This is a PStaticMesh with the ability to detect collision between itself and the world. Any type that is of PStaticMesh can be collided with, but only objects of type PCharacter can detect that collision.
4. **PObject::Plight [Base]** Derives from PObject. The PLight class is simply a container for other lights. If something lights, it derives from PlLght. Each light carries different statistics, for example, Radius, Strength, and DebugName. A PLight will not emit light but is used in all lighting types as a data center.
5. **PObject::PPointLight** Derives from PLight. A Point Light emits light in a circular radius from a single point. Its radius, intensity, and color can be customized, and many can be placed together in a scene. This is the most common light and simulates standard household/outdoors lights.
6. **PObject::PDirectionalLight** Derives from Plight. Directional Lights are sunlight-like lights that emit light in one direction. If a surface faces that direction, it will be lighted by the Directional Light; if a face is facing away from the directional source, it will not be lighted.

**II.** **Renderer**

**i. Renderer vs Environment** The renderer is responsible for receiving your requests from the Environment and ensuring that either the request is handled, or the failure is communicated to the Environment and end-user. The renderer takes all information you supply to the Environment through the Editor and renders it on-screen based on a bunch of settings from within the Engine.ini file. When the renderer boots up, it will load all settings and tell the Environment how to use them. The Renderer is also the layer that creates and displays the GUI for the Spore Editor!

**III.** **Environment**

1. **What is an Environment?** The Environment will communicate with the renderer for you. It also handles object creation, memory management, loading and saving, and many other generic world on goings. Think of the Environment as a middle man that takes what you create and ships it around the world for you; it eliminates any reliance on the renderer, so you don’t have to touch any graphics code if you don’t want to. There can only ever be one Environment, and it always holds a reference to each and every object created in the game world you currently have loaded.

**ii. Environment/Renderer States** Environment modes/states provide a way to quickly change debug settings to give you many ways to error-check development, while still giving you a way to disable these settings when playing if need be. See more about the states in Environment Pipeline. There are three modes to run in, and they are as follows:

* *DEBUG* Debug mode shows all object matrices, a world grid, and all lighting-object matrices. All Output Log messages are printed.
* *TEST* Test mode shows only object matrices. Only critical Output Log messages are printed.
* *SHIP* Ship mode effectively disables the environment debug features. No Output Log messages are printed. This cannot be enabled from the editor.

**iii. Saving and Loading** When you save a level, it is automatically saved in the "…/PolynGame/Levels/" directory. Levels in the Polyngine are saved as files with a ".plevel" file extension. This file-type is unique to Polyngine, and it stores all information the Environment needs to display an entire level properly. This includes PObjects, PCameras, PStaticMeshes, PLights, PPointLights, PDirectionalLights, and more.

**iv. Saving Breakdown** You can save the level you currently have loaded at any time by pressing “CTRL + S”, just be sure you are saving the right level to the right file. Saving a level consists of breaking the Environment down into its core component: WorldObjects. Saving your level stores each object and its information in a file. In this file, each individual line represents a new object. Note that this is not for game loading and saving, only to save and load editor levels that will be interacted with by the end user. Levels should always be saved in the “PolynGame/Assets/” directory. You are free to save it into subdirectories within “/Assets/”, but do not save outside of “/Assets/”.

**v. Loading Breakdown** You can load a level at any time by using “File -> Open Level”. Loading a file consists of reading in a saved “.plevel” file and creating a new object in the game world for each object that was saved. As it loads objects and creates them, it will gradually attach Children to their Parents, Create and Assign Controllers, and restore all object settings. When an object is created (if it has a Parent Object assigned), the engine will attempt to attach it to the Parent; if the parent is not yet created, the relationship is added to a list to be processed once all objects are loaded in and created. The reason the engine tries to attach before adding the relationship and defaulting to attaching in the end is for speed. By checking while we add them to the world, we can effectively reduce the number of loops we have to do at the end of each frame.

1. **Environment Pipeline** The Environment will keep a collection of everything that is currently loaded into the game world, and controls what data can be sent depending on the Render State. When the editor loads any level, it defaults to DEBUG state, which effectively tells the engine what and how to display information for debugging and editing. When the test button is pressed, the Environment Render State is set to TEST, which keeps many of the debug capabilities and customizations for editing, but displays it in a way more keen to how it will function and look in the final version. When a game is started in SHIP mode it will launch and only can only run the game, or close. Closing it will not open the editor or make any calls, it simply closes. All debug functionality is disabled in SHIP, including (but not limited to) Debug Display Grids, Matrices, and Lines, Output Log printing and reading, Level Editor, Object Inspector, Output Log window, and more. If SHIP mode is executed while using the editor, it will functionally disable the engine and restart the game using only the renderer and files. After exiting the game, you will be able to start the editor back up and the game will be back in DEBUG mode.
2. **Creating Objects** Never create an object by calling new() on it. The Environment is there to handle object memory for you, so always use the Environment as a middleman for creation and interaction with objects. Anytime you create an object, a pointer to the newly created object (the class will already be casted to the correct type [ex. PPointLight or PStaticMesh]) will be returned from the function call. You can save this pointer and use it to interact with the world and the object. Anything derived from PObject can be cast from a PObject to its class type with an error-checked dynamic cast. When creating any object using an Environment creation call, leaving the “DebugName” as default or “” will result in the object being given a random and unique name; the same result will happen if you try to name an object a name that another object already has. This random and unique name will always start with the class type (such a StaticMesh\_... for a PStaticMesh) and represent the object in the world.

**PEnvironment::CreateController()** – Create a PController and ready it for use by the engine. This function takes no parameters. Return created controller.

**PEnvironment::CreateObject(bVisible, DebugName)** – Create a PObject with the name “DebugName”, and set its visibility to “bVisible”. Return created object.

**PEnvironment::CreateCamera(FOV, bAssignInput, bSetActive, DebugName)** – Create a PCamera with the name “DebugName”. If “bAssignInput” is true, the creation process will assume a PController exists and is already attached to this object. The “FOV” is the field of view to initialize the camera with; this can be changed at any time. Return created Camera.

**PEnvironment::CreatePrimitive(Type, bVisible, DebugName, Parent, InScale)** – Create a PStaticMesh with the name “DebugName”, then load a primitive type’s (“Type”) vertex data into it and set its visibility to “bVisible”. The object will be scaled to “InScale” and attached to “Parent” object if it exists. Return created primitive static mesh.

**PEnvironment::CreateStaticMesh(ModelFilePath, DDSFilePath, bVisible, DebugName, Parent, InScale)** – All properties are the same except for “ModelFilePath” and “DDSFilePath”. The 3D model associated with this Static Mesh will be loaded from “ModelFilePath”, if the type is not a primitive, and the Texture will be loaded from “DDSFilePath”. Return created static mesh.

**PEnvironment::CreateDirectionalLight(Strength, Dir, Clr, DebugName)** – Creates a PDirectionalLight with the name “DebugName”, sets the intensity of the light source to “Strength”, the direction the light is coming from to “Dir”, and the color of the light source to “Clr”. Return created directional light.

**PEnvironment::CreatePointLight(Strength, Rad, Clr)** – Creates a PPointLight with the name “DebugName”, sets the intensity of the light source to “Strength”, radius of light emitted from the source to “Rad”, and the color of the light source to “Clr”. Return created point light.

**PEnvironment::DestroyObject(Obj)** – Destroys the object located at the memory for the passed in object pointer “Obj”. This function ensures memory cleanup by calling all necessary Destroy() and EndPlay() functions and cleanup helpers. If you do not call this function on an object before the renderer exits, it will automatically be cleared by the Environment.

1. **Using Controllers** Controllers are built to be a simple component that exists for an object to reference when needed. Because of this, to give an object a controller, you just need to call PossessController() on any PObject derived class (see more about PossessController() below). Once you call that function, given that the controller is valid, the object will have its “Controller” member set to the “InController” and be allowed input from the InputManager via the Controller.

**PObject::PossessController(InController, bActivate)** – “InController” is the controller that will be used for this object, and the “bActivate” parameter tells the object whether it should enable input or not. [ex. WorldObject->PossessController(CreateController(), true)] Once a controller has been possessed by a PObject type, that PObject can then check the controller for input events.

**IV. User Input**

1. **How is Input Used?** Input is read by Windows and must be communicated to the game engine to tell what actions the player wants to take. The pipeline to read input is critical to allow a true gameplay experience. In Polyngine, input is read and interpreted by two different components; the component that reads input is the PInputManager class, the component that interprets that input is the PController class. For input to be acted upon you must program the functionality into a PObject class and read the input from a Controller (that also reads that input from the PInputManager). For more information on PInputManager and PController read the rest of this section.
2. **PInputManager [Class]** The PInputManager is not to be used as a controller, or even created at all; it merely collects input, queues it up in a queue, then processes all input at the end of a frame. To ensure that all input fired is collected, the Window Handle will fire an event every time any input is triggered and add the input to a queue. The information queued up includes not only the key, but the state of that key that was inputted. Every frame, the entire input queue is processed in the order they were received. This allows the user to input multiple keys per frame and still process it even if there is not a frame between inputs. The processed input is stored in a bit structure that holds three bits (Input: Held, Held Last Frame, Just Released). The PController is the object that interprets the input queue.
3. **PController [Class]** A PController does not inherit from the PInputManager class, but it does use it to read the input information. The Controller differs from the Input Manager in the sense that the Controller is aware of the player that it controls, whose input was fired from the Input Manager, and what input is active. Calling the functions to test for input will ask the Input Manager if the key is pressed for that player index, then the Controller will communicate that information to the Object that is requesting it. To communicate input information to an object, you must first create a controller for the object (one per object), then Possess the controller with the object that will read the input. Once an object has possessed a controller, you can then query the Controller through the class variable that holds it. To disable input temporarily, you can call the EnableInput(false) function on the object, to disable the controller permanently, you can simply destroy the controller.

**IV. Materials**

1. **What is a Material?** Materials are your way of dynamically interacting with the shader pipeline. When you change values on a material, it will be reflected in real-time in-engine and during gameplay and can be altered on-the-go! With this system, you can code animations and dynamic affects into the shaders of an object without requiring any graphics code to be added. Materials come standard in the PStaticMesh class and the PSkeletalMesh class and have default values that cause the shader to display exactly what the textures tell it to. To access the Material of an object you can either select it in the Level Editor or use the PStaticMesh->Material command and directly manipulate its components.
2. **Manipulating Shaders** Manipulating a material is as easy as changing the variable of the component you would like to alter. For example, PStaticMesh->Material.Specular = { 1, 1, 1, 1 }; will set the specular to maximum for the entire model, while , PStaticMesh->Material.Specular = { 0.5f, 0.5f, 0.5f, 0.5f }; will set the specular value to half for the model and use the specular texture to decipher what should be increased after.