WSU Cleanroom VR Project

2021-2022 Senior Capstone

Team 10

Brandon Huang, Samantha DeShazer

Gema Hernandez, Ryan Paulos

Laura Messick, Trey Fontaine

**Contents**

Running the Project on the Oculus Quest 2…………………………………………………………………………………………………………………….3

Prerequisites………………………………………………………………………………………………………………………………………………………3

Procedure…………………………………………………………………………………………………………………………………………………………..5

Sputter Process……………………………………………………………………………………………………………………………………………………………….6

Blender 3D modeling tips………………………………………………………………………………………………………………………………………………..7

Button Grab Components……………………………………………………………………………………………………………………………………………….7

How do I use them?.................................................................................................................................................7

What if I want to modify them? ……………………………………………………………………………………………………………………….14

What if I want to change key bindings? ……………………………………………………………………………………………………………14

Unresolved Issues ……………………………………………………………………………………………………………………………………………15

Configuring an Unreal project for Oculus App Labs……………………………………………………………………………………………………….16

Before Configuring an Unreal Project ………………………………………………………………………………………………………………16

Configuration Settings………………………………………………………………………………………………………………………………………16

Uploading to Oculus App Labs………………………………………………………………………………………………………………………….17

Video and website Links…………………….…………………………………………………………………………………………………………………………17

Our documentation of the original VR project………………………………………………………………………………………………………………18

Disclaimer………………………………………………………………………………………………………………………………………………………..18

The Project……………………………………………………………………………………………………………………………………………………….18

The Code………………………………………………………………………………………………………………………………………………………….18

Header Files…………………………………………………………………………………………………………………………………………18

Class Files…………………………………………………………………………………………………………………………………………….18

Blueprints…………………………………………………………………………………………………………………………………………………………18

Levels……………………………………………………………………………………………………………………………………………………………….19

Common Functions…………………………………………………………………………………………………………………………………………….19

KeyboardPawn……………………………………………………………………………………………………………………………………………………19

Buttons………………………………………………………………………………………………………………………………………………………………20

KeyboardPawn, Button, Machine interplay………………………………………………………………………………………………………..21

Wafer Production……………………………………………………………………………………………………………………………………………….21

Equipment………………………………………………………………………………………………………………………………………………………….21

**Running the Project on the Oculus Quest 2**

**Prerequisites**

1. Download and open android studio: (Need this to automatically install some things we need like the android sdk)

<https://developer.android.com/studio>

1. Download command line tools in Android Studio: (We need these to setup android development for unreal)
   1. Create or open any project
   2. Go to file>settings, then Appearance & Behavior > System Settings > Android SDK
   3. Go to the SDK Tools tab and check android SDK Command-line Tools (latest), hit okay and let it download.
2. Download Flutter: https://flutter.dev/docs/get-started/install/windows#run-flutter-doctor (we need this to accept sdk licenses)
3. Open a command prompt in the bin folder at the file location where you installed flutter and type flutter doctor, changing the command prompt to the directory:<YourDirectory>/flutter/bin). It should say there are some issues, then type: flutter doctor –android-licences. And accept all the unaccepted licences. \*\*\* You can also add flutter to your path variables in windows, but we only need to use flutter for this step.\*\*\*
4. Now to go to your UE4 install directory, and locate the "SetupAndroid.bat" in Engine/Extras/Android of your UE4 Install directory.
5. Now open up the SetupAndroid.bat file in a text editor like notepad++ and edit line 61 which says:0 set SDKMANAGER=%STUDIO\_SDK\_PATH%\[HERE IS A WRONG DIRECTORY] \*\*\* You can also get around this by modifying your path to prioritize the command line tools, but its much faster to just edit the script\*\*\*
6. Change the line to this exactly: set SDKMANAGER=%STUDIO\_SDK\_PATH%\cmdline-tools\latest\bin\sdkmanager.bat. Once the line is changed, run the SetupAndroid.bat and let it finish.
7. Go to this file location for your Android tools section: \*\*\*To get to your app data folder, type "run" in the search bar in windows and type %appdata% into the run prompt.\*\*\* (mine is here: C:\Users\user\AppData\Local\Android\Sdk\build-tools\31.0.0)
8. Locate the file d8.bat, and if there isn't a file called dx.bat, rename d8.bat to dx.bat
9. Go to the lib folder in this same directory. Locate d8.jar and if there isn't a file calld dx.jar, rename d8.jar to dx.jar. \*\*\* Steps 8 through 10 won't always be necessary, They were for me but not for others. It's only necessary if in the directory there isn't the files: dx.bat and lib/dx.jar\*\*\*
10. Ensure adb is installed (should be already at this point, you may need to run the SetupAndroid.bat from step 5)
    1. open a command prompt and type the command: adb
    2. This should display some text about adb.
    3. If it isn't installed, you can get it here (with installation guide: <https://www.xda-developers.com/install-adb-windows-macos-linux/>)
11. THESE STEPS ARE NECESSARY IF THE OCULUS ISN'T ALREADY IN DEVELOPER MODE:
    1. Setup an oculus developer account
    2. Link the account to the headset (if an account is already logged in, you must factory reset)
    3. Using the oculus app you used to setup the headset, enable developer mode on the headset.

\*\*\*I didn't personally do these steps, but Ryan and Laura did. They can answer questions on details. Here are some guides: Text guide: <https://learn.adafruit.com/sideloading-on-oculus-quest/enable-developer-mode>. video guide: <https://www.youtube.com/watch?v=RoIXxIfRNTw> . The video describes the process from 2:50 to 5:00\*\*\*

1. Plug the headset into the computer and put the headset on after developer mode is activated on the headset. A popup in the headset should appear asking if the user would like to enable USB debugging and the computers RSA key fingerprint. Select to remember the computer for later and yes. The next popup in the headset should ask if you would like to allow the computer to access files, say yes to this one as well. \*\*\* NOTE: IF THE FIRST POPUP IN THE HEADSET SAYS A DIFFERENT MESSAGE THAN ENABLE USB DEBUGGING AND THE RSA KEY, YOU DON'T HAVE DEVELOPER MODE ENABLED ON THE HEADSET AND NEED TO GO TO STEP 12.\*\*\*
2. Once USB debugging is enabled, UE4 should be able to recognize the device while plugged in. You can check everything is setup properly by typing the command in a command prompt: adb devices. This should show one device connected on adb, the headset. If its empty, developer mode probably isn't active or you didn't allow usb debugging. UE4 should also show the headset in the launch dropdown tab.
3. We still need to get a jdk, specifically Java SE 8. If you don't already have it you either need to:
   1. If you have an Oracle account/acess, here: <https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html>
   2. Otherwise we can use an OpenJDK build (what I'm using): <https://www.openlogic.com/openjdk-downloads>
   3. If you use the openJDK, unzip it in this directory: C:/Program Files/Java
4. Open a project in unreal and configure it for VR. Follow the steps detailed here: <https://developer.oculus.com/documentation/unreal/unreal-quick-start-guide-quest/>. UPDATE FOR ANDROID BUILD TOOLS VERSION 22, I had to set the minimum sdk version to 22, and I'm pretty sure as it gets updates.

- We need to set the minimum sdk version to that (Ex: version 23, set minimum sdk to 23).

- Make sure the project is set for tablet/mobile, graphics are 2D/3D scalable.

- Make sure you follow all the configuration steps in the "Configure the Project for Oculus Development" section.

- When setting the paths in the project to the sdk,jdk and ndk, here's where mine are set to, and I used default locations:

Android sdk (installed when we installed android studio):

C:/Users/USER\_NAME/AppData/Local/Android/Sdk

Android ndk (installed when we ran that bat script in the UE4 folder): C:/Users/USER\_NAME/AppData/Local/Android/Sdk/ndk/21.4.7075529

jdk (MUST use Java SE 8 here. Other jdk's such as jdk 14 will not work): C:/Program Files/Java/openlogic-openjdk-8u262-b10-win-64

**Procedure**

1. If you don’t already have the project open in Unreal: In he project folder, navigate to VR-Project/CleanVR. Double-click CleanVR.uproject to open the project in Unreal Engine. Respond to the prompts to allow Visual Studio to compile the project code if necessary.

Graphical user interface, table

Description automatically generated

1. Launching the project should display the following view.

A screenshot of a computer

Description automatically generated with medium confidence

1. There is only one level in this project however multiple levels can be made. (Named “TestLevel” here)
2. Now from the launch dropdown in unreal, select to launch from the headset, it will take a while to build everything the first time.

Graphical user interface, website

Description automatically generated

1. After some time, the app should be running on the headset! Unreal will still say building while its running, if you open the output log in unreal, after this line displays:

LogPlayLevel: \*\*\*\*\*\*\*\*\*\* RUN COMMAND COMPLETED \*\*\*\*\*\*\*\*\*\*

LogPlayLevel: BUILD SUCCESSFUL

The app should be running on the headset.

**Sputter Process**

1. Place wafer onto the chuck.
2. Open the door to the machine.
3. Place chuck inside the sputter machine. Attaching it to the square disk at the top.
4. Close the door.
5. Press the pressure button to depressurize the machine.
6. Press the spin button for rotation.
7. Set deposition level.
8. Press the power button to turn on the machine.
9. Once depressurized the machine starts and a timer counts down. Plasma should be glowing here (square disk the chuck is attached to instead of the plasma disks).
10. After the countdown is finished, turn off the power by pressing the power button.
11. Press the pressure button to pressurize.
12. Turn the rotation off.
13. After it is pressurized, it’s no longer rotating, and the power is off the door can be opened.
14. Remove the chuck and notice that the wafer has changed color. It is now metallic and shiny.

**Blender 3D modeling tips**

* Make 3D models proportional and to size in real life, otherwise this causes issues when importing to Unreal. For example, if you want an object like a wafer that is 3 inches in real life, make it 3 inches in blender. Or a machine that is 5 feet tall, make it 5 feet in blender.
* Always be careful about proportions when joining or separating pieces, this can be tricky.
* Don’t worry about the color of your model, textures and materials will be added in Unreal. Just make sure if there will be more than one material on a model that that is reflected in blender. Let’s say a model will have a wood and a metal texture, in blender you could make all the wood sides red and all the metal sides blue.
* Export as fbx files, the fbx files are imported to Unreal.
* Anything that moves/separates in Unreal should be in different blender files. Like a door on a machine, the door is a separate blender file than the machine.

**Button Grab Components**

To whoever is reading this guide,

I wanted to quickly throw together this guide on how to use my grab and button component blueprints in Unreal. I spent quite a bit of time on these at the beginning of the semester as a baseline system, and here’s some documentation on how to use them. I hope this is helpful and Good luck in using Unreal. This project was my first time using it, and man, was it a learning curve.

-Brandon Huang

**How do I use them?**

These components are intended to be attached to Static Mesh Actors or Static Mesh Components. Grab’s have 4 types, None, Free Snap, and Pivot. The item it’s attached functions differently depending on the type it’s set to. Buttons have different types as well, one for each possible functionality of the button. The general process is to attach the component to a static mesh, then set the type. Grab’s are bound to the grab buttons on the quest controllers, and buttons are bound to the trigger presses.

*Grab component:*

The VRPawn blueprint has an event that triggers on the grab button presses. It then searches for the nearest grab component and checks if your hands are overlapping with the object. If it is, it stores a reference to the object and calls the TryGrab function in the grab component. Depending on the grab type set on the component, it will do one of 4 things:

1. None: Nothing will happen, this is an empty grab.
2. Free: The object will attach to the player’s hand at the point where they grabbed it.
3. Snap: The object will attach to the player’s hand at the root of the object.
4. Pivot: The object will stay in place, but pivot with the player’s hand along the XZ plane. (Like opening a door)

How to use (In a static mesh actor):

1. Place your static mesh in the scene
2. Select the static mesh actor by clicking it on the scene or in the world outliner
3. Ensure that Physics is enabled for the object, it won’t work otherwise: Graphical user interface

   Description automatically generated
4. Select add component, and add the Grab Component. The component should be attached to the static mesh as shown below.Graphical user interface, text, application

   Description automatically generated
5. Select the newly added component and under default, set the grab type.A screenshot of a computer

   Description automatically generated
6. If we're using a pivot grab, set the other three fields as well. They don’t change anything unless a pivot grab type is set. They do the following:
   1. Max Rotation Angle: The maximum angle the player can rotate the object to.
   2. Min Rotation Angle: The minimum angle the player can rotate the object to.
   3. Pivot Offset: In case your mesh is default rotated in not the positive XZ direction, you can use this to offset the angle between the player’s hand and the object.
7. That’s it, you should be able to grab them in the scene.

How to use (In a blueprint):

Just like attaching the component to Static Mesh Actors in the scene, we can attach the blueprint to Static Mesh Components in a blueprint. Just ensure that the parent component of the grab component is the mesh you want to be grabbable. It should look like this:

Text

Description automatically generated

You also must ensure that the relative location of the component to the mesh is (0,0,0) to ensure it works properly and that physics is enabled for the static mesh component it’s attached to.

Graphical user interface, application

Description automatically generated

WARNING: If you have a blueprint with both grab and button components somewhere in the hierarchy, you MUST place a grab component set to None on every mesh with a button component, and a button component set to None on every mesh with a grab component. Explained more in the Unresolved Issues section.

*Button Component:*

These components are purely meant to be used in blueprints since they need to have contextual functionality when they’re pressed. They can execute any arbitrary code you specify but unfortunately have their code split between 3 different blueprints. They are activated when a player presses the trigger when their hands are intersecting with an object which has a button component attached to it. When the player presses the trigger, an event is triggered in the VRPawn blueprint which searches for the nearest button component to the controller which pressed it. If the player’s hand is overlapping with the parent mesh of the component, the VRPawn calls the TryPress function of the ButtonComponent. This function checks what type of button it is, gets a reference to the parent blueprint the button is supposed to be on, and attempts to cast the parent object to the correct blueprint. If it succeeds, it calls the appropriate internal function of the parent object blueprint, doing whatever it needs to do.

1. Open the blueprint where the button is going to be, and add a function that you want to be executed on button press. You can make it do whatever you want.Graphical user interface, text, application

   Description automatically generated
2. Attach the button component to the mesh you want to be the button itself. Do this by clicking on the mesh and selecting Add Component, and selecting the Button Component. It should look something like below:

Graphical user interface, text

Description automatically generated

1. Ensure the relative location, rotation, and scale of the component to the attached mesh is neutral across the board:Graphical user interface, application

   Description automatically generated
2. Open the TriggerType Enumeration type in the Content/Blueprints/ButtonCompenent folder and add a new type:Graphical user interface, application

   Description automatically generated
3. Set the Display Name to what you want your new button to be called and add a description if you want.
4. Now open the ButtonComponent blueprint in the same folder and open the TryPress function. Your new button type should be at the bottom of the Switch on TriggerType node: Diagram

   Description automatically generated
5. Now drag off the new pin and attempt a cast to the blueprint where your button is attached to. In my example, I’m casting to my Quizboard blueprint. You want to set the object field of the cast to the return value of the Get Owner function if your calling a function inside the blueprint were casting to.A screenshot of a computer

   Description automatically generated with medium confidence
6. Drag off the As <Object Name> Node on the Cast node and search up the function we made to run on the button press of that blueprint. Connect the execution order and everything is hooked up!

Note: Technically you don’t have to do a cast and function call of the blueprint, you can just put all the code in the ButtonComponent blueprint, but the TryPress function is already quite cluttered and that would make it even worse.

WARNING: If you have a blueprint with both grab and button components somewhere in the hierarchy, you MUST place a grab component set to None on every mesh with a button component, and a button component set to None on every mesh with a grab component. Explained more in the Unresolved Issues section.

**What if I want to modify them?**

The components and interactions with them are in three places, the VRPawn controller pawn, the component’s blueprints themselves, and their respective enumeration types. If you want to modify them you’re going to have to look into them.

**What if I want to change key bindings?**

Binding events to key presses is done through the Edit->Project Settings ->Input->Bindings in the editor. Here you can add bind and add events to certain key presses.

Graphical user interface, text

Description automatically generated

If you just want to change the binding for the event, do it from here.

If you want to change the event itself, go to the VRPawn blueprint. Here you can see each of the events and modify what they do. You also add events here once adding them to the Bindings in the project settings.

A screenshot of a video game

Description automatically generated

**Unresolved Issues:**

There is one major bug with these components. I didn’t realize it existed until too late in the semester. If a blueprint contains both Button and Grab components on different static mesh components within the hierarchy, the mesh with a button component will associate with the nearest grab component and the mesh with a grab component will associate with the nearest button component. This means you can press or grab things from the wrong mesh. This is because when the events trigger in the VRPawn, and it searches for the nearest grab/button component, they use the same trace channel, so a component grab component’s parent mesh will be found when searching for a button component, which it will then associate with the nearest button component. This is true for the opposite way around as well. This could be fixed by setting up separate trace channels for each component type, but that would be a very large undertaking and I didn’t have time for it. Instead, my workaround is to attach a blank/None type grab and button component to each mesh that has the opposite type. This means that when the Pawn searches for the nearest component, it will instead associate it with the None typed component instead and simply do nothing. This is a workaround and not a true fix, however, but fixing the systems would have taken some time I didn’t have.

The workaround where I have a None typed component attached: Graphical user interface, application

Description automatically generated

**Configuring an Unreal project for Oculus App Labs**

**Before Configuring an Unreal Project**

* Download flutter sdk from: <https://docs.flutter.dev/get-started/install/windows#run-flutter-doctor>

Follow directions to update your path if necessary.

Run: flutter doctor

Follow the section on downloading android studio (3 steps).

Run: flutter doctor –android-licenses.

* Install Visual Studios community edition. While downloading make sure you check the option for “Desktop development with C++”
* Set up an Oculus developer account if you don’t already have one. Things you’ll need: e-mail, Facebook account, a phone number and credit card attached to that Facebook account. Create a new Unreal project and just give it a name for now.

**Configuration Settings**

**Note:** Do Not try to change the AndroidManifest.xml file directly. This is compiled during packaging and will not save your changes.

* Edit->Project Settings->Plugins->OculusVR
  + Make sure under Mobile that Focus Aware is checked (Probably already is)
* Edit->Project Settings->Platforms->Android
  + Android Package Name… change to “com.[PROJECT]” (PROJECT here will change to your project name automatically)
  + If there is a red bar saying “Project is not configured for the Android platform, click on configure now.
  + Change Minimum SDK Version to at least 23 (Note: I chose to set it to 24)
  + Store Version should start at 1, then increment every time a build is uploaded to Oculus app lab.
  + Install Location should be set to auto
  + Under Build-> uncheck Support armv7 and check Support arm64
  + Orientation should be just landscape, not sensor landscape
* Go to your project folder->Build->Android:
  + You should see res and src folders here. Create a text file named: “ManifestRequirementsOverride.txt”. In this text file:

“<uses-sdk android:minSdkVersion = "24" android:targetSdkVersion="28" />

<uses-feature android:glEsVersion="0x00020000" android:required="true" />

<meta-data android:name="com.oculus.vr.focusaware" android:value="true"/>

<uses-feature android:name="android.hardware.vr.headtracking" android:version="1" android:required="true" />

android.permission.INTERNET

android.permission.ACCESS\_NETWORK\_STATE

com.android.vending.CHECK\_LICENSE

android.permission.INTERNET

android.permission.WAKE\_LOCK

android.permission.ACCESS\_NETWORK\_STATE

android.permission.ACCESS\_WIFI\_STATE

android.permission.VIBRATE

com.android.vending.CHECK\_LICENSE

com.android.vending.BILLING

android.permission.CAMERA

android.permission.READ\_PHONE\_STATE

android.permission.GET\_ACCOUNTS

android.permission.MODIFY\_AUDIO\_SETTINGS”

* Go to: <https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Mobile/Android/DistributionSigning/>

Follow directions on creating a key. Make sure you find the correct keytool.exe. It should be in Android Studio \jre\bin folder. (For example my full path was C:\Program Files\Android\Android Studio\jre\bin). Make sure to run cmd prompt in admin mode to avoid error message. Keystore password I set: FabVR22.

**Uploading to Oculus App Labs**

* In Unreal: File->Package Project->Android->Android(ASTC)

Make sure packaging is successful.

* In Oculus Developer Hub (app):
  + Click on app distribution
  + Click on your project. (This should have been set up already via Oculus developer dashboard, website)
  + Click on upload for either alpha, beta, RC, or Production.
  + Select the .apk file which is located inside of the Android\_ASTC folder which is wherever you saved during the packaging step.
* Your build should now be showing on both the Oculus app and website!

**Video and Website Links**

* Starting at 40 seconds in, WSU cleanroom, removing a wafer from the sputter machine:
  + <https://youtu.be/g9gQS6_F1s0>
* How to Submit an App for App Lab:
  + <https://developer.oculus.com/blog/introducing-app-lab-a-new-way-to-distribute-oculus-quest-apps/>
* Converting UE projects to different versions of UE (UE5 just came out at the end of this project):
  + <https://docs.unrealengine.com/4.27/en-US/Basics/Projects/UIProjectConversion/>

**Our documentation of the original VR project**

**Disclaimer**

The contents of this document reflect our best understanding of the legacy project. We (21-22 team) inherited this project from the year before. It does not run on the Oculus headset. Reviewing the implementation details may help in understanding Unreal.

**The Project**

The legacy project is largely built from C++ source code files. Each code contains a class that are used as base classes by the Unreal engine, inheriting from these source code files to create “Blueprints”. There are pre-made base classes available in Unreal as well. **Project blueprints are located in Content/Items**. We highly recommend watching some tutorials on Youtube to become acquainted with the basics of the editor. This one is a good intro: <https://www.youtube.com/c/MakeGamesWithKatie> . Also reference the official Unreal documentation: <https://docs.unrealengine.com/4.27/en-US/API/> .

**Note: All functionality can be done purely in Unreal. Source code files are essentially just imported and then optionally modified further in the Unreal editor.**

**The Code**

Located in the Source directory. 'Private' contains the class files while 'Public' contains the header files.

Header Files:

The .h files are where class variables are declared, which includes UPROPERTY variables. These are property specifiers specific to Unreal Engine.

**Note: Changing (adding, updating, or removing) UPROPERTY values in a .cpp class file will probably corrupt any blueprint based on that .cpp file. If corrupted, the blueprint is now worthless. A new blueprint will need to be created.**

Class Files:

The .cpp files are all class files, each corresponding in an in-world structure. This includes tangible objects, like Wafer.cpp corresponding to an in-engine wafer, and Jug.cpp corresponding to an inengine jug. Intangibles, like RulesSetFive.cpp and RulesSetFour.cpp, are also implemented (these ones in particular tracks the progress of the wafer through the production process, updating the in-game clipboard).

These .cpp files are imported into Unreal Engine by “File → new C++ class”. Make sure all “Show All Classes” is checked. The files should be available to load.

**The Blueprints**

A blueprint has “components” in the top left of the blueprint window. Clicking through each component opens up the “detail” window in the right side, allowing the variables to be set. For example, the SpinMachineBP blueprint has a Spin Stage component, with the UPROPERTY declared in the SpinMachine.h (the component hierarchy being declared in the constructor method in the SpinMachine.cpp). But the actual assets those variables refer to are assigned in Unreal. In this case that would be the Spinner\_Platform static mesh.

Really, each blueprint is a class definition while each object (“actor”) in the game world is an instance of one of those classes.

**Levels**

In the Unreal editor go to the Content/Maps/ directory and double-click “Lab” to load the proper level where the machine are located.

**Common Functions**

**::BeginPlay()**

This method is called upon object spawn.

**::Tick(float DeltaTime)**

This method is run every time the engine renders a new frame.

**::Interact(AActor\* instigator, Fvector location)**

Many in-game objects use IInteract (from interact.cpp) as a base class, making this method available. These are interactions performed with the 'F' key. Encompasses all actions except for button presses. When using the Keyboard interface the objects must be held first (grab by pressing the space key). This method is called from the KeyboardPawn class (which is what you are when playing with keyboard controls).

**UkismetSystemLibrary::SphereOverlapActors..**

The go-to method for checking for the presence of another world object. This method is used to produce a list of all world “Actors” of a given class within a given radius.

**KeyboardPawn**

“You” when playing the legacy project using keyboard controls. The “cursor” is a red sphere that, in conjunction with the above mentioned SphereOverlapActors call, checks for all world objects of the types specified in the objectTypes list. KeyboardPawn attempts to cast each element to an AItem, toggling the grabbed item's grabbed state if the cast is successful. Press 'space' to grab items. Once grabbed, press 'F' to call its ::Interact() method.

KeyboardPawn has a different mechanism for buttons. In the ::Tick() method a LineTraceSingleChannel() call is made to check for any button objects on the path represented by the “cursor”. This is then saved to the instance variable SelectedButton. This value is set to NULL if the path then moves away from a button. Buttons are pressed by using the 'E' key while in the cursor's path.

**Buttons**

Derived from AbuttonPress.cpp. All machine buttons are defined in the header files as AbuttonPress\*. Note that these are pointers. The buttons themselves need to be created within Unreal and then assigned to their respective instance variables. The following screenshot shows where this is done.

Graphical user interface, application

Description automatically generated

**Note:** The two objects must exist in the same sub-level for this to work. The labs level was divided into sub-levels in the hopes of enabling multiple people to work on the project at the same time. Managing sub-levels: <https://docs.unrealengine.com/4.27/en-US/Basics/Levels/LevelsWindow/>

**KeyboardPawn, Button, Machine interplay**

Once the buttons are assigned to the AButtonPress pointers and the “game” starts, the machines launch their ::BeginPlay() methods. There they check that the pointers are not null. If so, the machine registers itself with the buttons' interactWith' variable. When a button is brought within the path of the KeyboardPawn's cursor and made the SelectedButton, pressing the 'E' key triggers a

KeyboardPawn → AbuttonPress\* →

cascade to the machine's IInteract method. This method will check which button invoked it and continue program execution accordingly.

**Wafer Production**

The various machines update modify the wafer as it undergoes the development process. Some of the logic for the appearance change is tied to the dynamic “material” assigned to the wafer. Go to

Content/Assets/Wafer/, right-click WaferMT->Edit.

The implementation toggles the visibility of different states using the Alpha value from the Wafer::Tick() method. There are additional materials that are toggled throughout the development process.

**Equipment**

**Tweezers:** Uses SphereOverlapActors to grab nearby qualifying objects.

**Beaker:** Operates through object collsion by setting onComponentHit. The beaker then checks what type of object it has collided with and updates itself and Wafer accordingly. Updates wafer material.

**Airwand:** Uses line tracing to determine if a wafer is affected by the stream. Updates wafer material.

**SpinMachine:** Uses SphereOverlapActors to locate the Wafer. Updates wafer material.

**Hot Plate:** Doesn't actually interact with the wafer object.

**MaskAligner:** Doesn't actually interact with the wafer object.