Prof. G. Zachmann
T. Hudcovic (hudo@uni-bremen.de)
R. Fischer

University of Bremen
School of Computer Science
CGVR Group
November 28, 2024

Winter Semester 2024/25

Assignment on Virtual Reality and Physically-Based-Simulation - Sheet 3

Due Date December 09, 2024

Exercise 1 (Laggy Jump & Run, *2 Credits*)

in this exercise we will test the influence of latency, to our performance in computer games. From the CGVR website (http://cgvr.cs.uni-bremen.de/teaching/vr/index.shtml), you can download a simple jump and run game ("lag_jump.zip") with adjustable input delay. The goal is to complete the course from start to finish, as fast as possible and without falling down. Please be advised that this has been tested against Unreal Engine version 5.3.2.

Note: We strongly recommend using that version. However, if you choose to use a newer version, we highly recommend using the exact recommended version (not a newer one!) of Visual Studio (and the MSVC) as noted here: [1], as using Visual Studio 17.12.X (the current version) will cause errors with NuGet-Packages in combination with Unreal Engine 5.4.X+, with the suggested solution being to manually downgrade the Visual Studio version to match the recommended ones for each respective Unreal Engine version.

- Each member of your group should complete the game with at least three different delays.
- Write down the number of starts needed for the first successful run, and the time for the fastest completion. Do this for each delay (note your delay) and for each member of your group.
- Plot and briefly discuss the results. For plotting, you can use any tool you like. Choose adequate diagram styles that make sense for the data you want show! Briefly describe the results and include the plots in your submission PDF.

[1] https://dev.epicgames.com/documentation/en-us/unreal-engine/setting-up-visual-studio-development-environment-for-cplusplus-projects-in-unreal-engine

Exercise 2 (Unreal Engine: Grab/Pick Up Implementation Analysis, *2 Credits*)

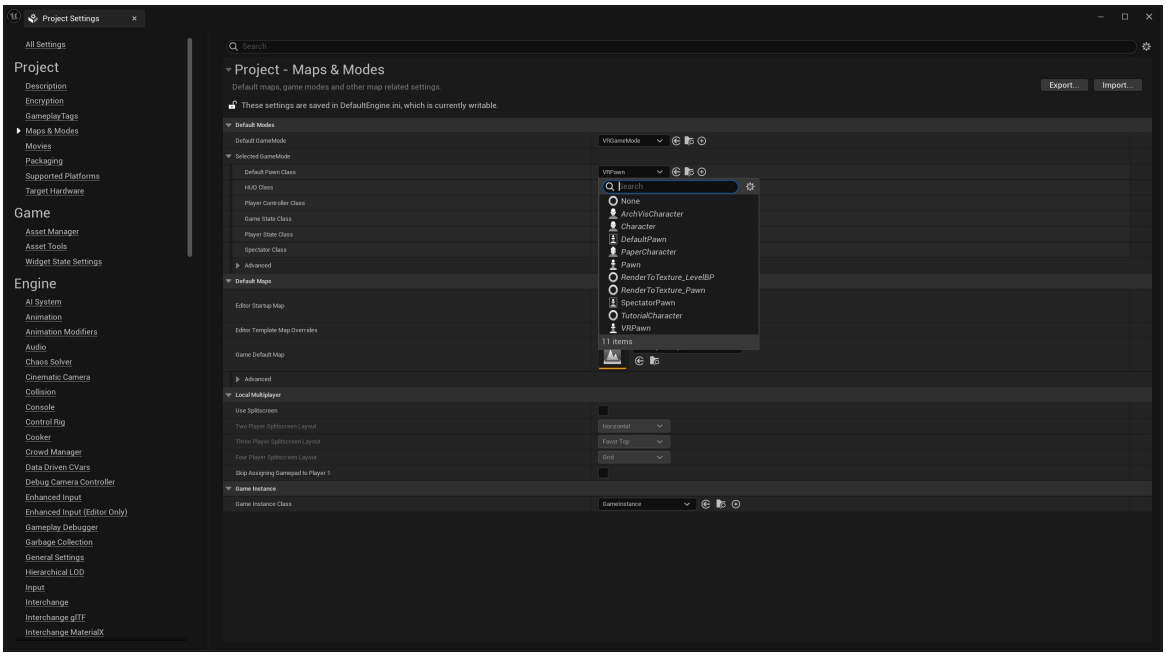

Figure 1: You can change the game mode and the spawned Pawn in the project settings.

Note: For this exercise, you don't really need a headset/HMD but it can be helpful (and fun). You can switch between game modes and/or Pawns. If you prefer for testing/development purposes to just have a regular, non-VR Pawn (it is called the *DefaultPawn*), you can change it in the project settings (see 1). This can be helpful for exercise 3.

Open a new project in Unreal Engine and select the VR template (no starter content). If you launch the game in VR (you might have to select the mode), the *PlayerStart*-Actor, which has already been placed in the level, will spawn a Pawn according to the selected game mode; in this case a VR-Pawn (unless you have changed it). You can play around a bit in the level. Try to pick up some of the objects on the table.

- Inspect the Blueprint(s) of the VR-Pawn by either clicking on it in the Actor list in the Unreal Editor during a running game or by searching for the VR-Pawn in the content browser (use search from the project root). Describe which parts of the Blueprint are relevant for picking up/grabbing an object and how that functionality is implemented.
- Inspect the Actors that can be picked up (like the cubes on the table). Why can they be picked up? And why/how are they included in Unreal's physics solver (take a look at the details panel of a grabbable object)?

Exercise 3 (Unreal Engine: C++ & LOD, *8 Credits*)

In this exercise you create an Actor in C++, inherit from it via Blueprints, and implement a LOD system for it. First, add a new C++ Actor to Unreal's VR template project from exercise 2. You

can do so under Tools → New C++ Class.. → Actor. This part [2] from the Unreal documentation seems especially useful to get started with C++, while this part [3] is the official C++-API documentation.

Don't forget to close the editor and build the project from the open Visual Studio instance.

- Our Actor should have a *UStaticMeshComponent* pointer as root component. Components can be created in the constructor via the *CreateDefaultSubobject* function. E.g.

  ```
  UStaticMeshComponent* MyStaticMesh =
      CreateDefaultSubobject<UStaticMeshComponent>(TEXT("My Box"));
  ```

  Don't forget to declare your class variables/components in the header file as well! One could then also add a mesh to it using something like

  ```
  static ConstructorHelpers::FObjectFinder<UStaticMesh>
      box_mesh(TEXT("StaticMesh'/Game/Box.Box'"));

  MyStaticMesh->SetStaticMesh(box_mesh.Object);
  ```

  The last line sets the mesh. Note that the type of the input argument is a **UStaticMesh*.
- For the LOD system, add 3 *UStaticMesh* pointer as public variables, as well as 2 float variables which act as thresholds.
- The Actor should be inheritable via Blueprints, so you have to add special Unreal macros and specifiers in the header file. This page [4] might be useful. Make sure all variables are marked as *UPROPERTY* with the *EditAnywhere* and *BlueprintReadWrite* specifiers. You may also group them into a category.
- Now, create a Blueprint Actor inheriting from the custom C++ Actor. At the *BeginPlay* event, one of the inherited meshes should be set as default mesh for the Actor's mesh component (use *SetStaticMesh*).
- On the CGVR course web site you find a zip file with 3 mesh assets you can use. You have to extract them into the content folder of the project and import them via Unreal. If you have the editor open already, a prompt in the lower right corner may appear to ask about importing the assets.
- Add one instance of the custom Actor to the scene and assign the 3 meshes to the corresponding properties via the editor.
- Implement a dynamic LOD selection strategy for the custom Actor, which depends on the estimated size on the screen. A rough estimate for the screen size/area is enough. The smaller the object is on the screen, the simpler the selected mesh should be. Use the inherited thresholds to switch the meshes in the mesh component. There are multiple approaches for this, you may get the mesh component's world space bounds, project them onto the screen, and compare its area to the one from the viewport. Useful Blueprint nodes are *GetComponentBounds*, *ProjectWorldToScreen*, *GetViewportSize*, and *GetAllActorsOfClass* to get a reference to the *PlayerController*. Note that the projected screen coordinates could be beyond the actual viewport, so you should clamp them or account otherwise for it. If the Actor is completely outside the viewport, no LOD changes should occur, and if only partially inside the viewport, only this part should be considered. Once again, the Unreal documentation can be useful to have at hand, e.g.: https://docs.unrealengine.com/5.3/en-US/BlueprintAPI/HUD/GetViewportSize/
- Remember the grababble Actors already lying around (exercise 2)? Make your LOD-Actor grabbable. If you throw/move it around, the LOD should change according to the Actor's size on the screen.

[2] https://docs.unrealengine.com/5.3/en-US/unreal-engine-cpp-programming-tutorials/
[3] https://docs.unrealengine.com/5.3/en-US/API/
[4] https://docs.unrealengine.com/5.3/en-US/cpp-and-blueprints-example/