Medienprojekt:
Passanten in Unity
Documentation
Stephan Brandt, 959966

Introduction:

The task onhand was to find a way to efficiently render AI game objects on a game, so that the performance is high even when there are many bypassers circulating our game, and therefore GPU usage is kept low and FPS (frames per second) are kept as high as possible at all times. In other words, to find a way to render only what is important to render.

This paper will explain the structure of the code, the usage of the different methods implemented, the insights on the development process (train of thought, difficulties experienced, bugs), and explain some added functionalities.

How to start the proyect on Unity3D:

A standalone .exe application is contained on the folder for a direct play-testing input without needing to install the proyect.

Firstable is important to take in note that this proyect is only going to work on Unity version 2017.2.1f1 and onwards, older versions won't work since Unity isn't backwards compatible.

1. Decompress the zip file. There should be a folder named "UnityProjectStephanBrandt", open this folder in Unity.

2. Open Scene1 on the Proyect view. At this point all the GameObjects should have appeared.
3. If for some reason the NavMesh is not activated, go to Window -> Navigation.
4. To set the NavMesh to work, click on the "Terrain" Game Object on the hierarchy. Next to the Inspector tab and Services there should now be a "Navigation" tab, go to Navigation-> Bake and press the "Bake" button.
5. A blue NavMesh should have appeared after a couple of seconds on the scene showing the walkable areas for our AI.

1) Brief structure of the project

1. *Scene structure and game objects:*
   Obstacles: Only there to test the AI movement and avoidance of static objects.
   Terrain: Simple terrain.
   Main Camera: Main camera of the game, contains a script to follow the player.
   Character: Our character.
   RadiusRenderer: Sphere without mesh used to deline the zone on which the AI will be either deactivated or activated.
   AIManager: Empty game object carrying a script where the logic of the game is contained.
   AICapsule: Prefab of our AI containing a basic AI movement script. (Prefab found on the Scripts folder).
   Top-down Camera: This is our second camera and it's main purpose is to serve as a mini-map looking from above.
   Canvas: Canvas containing some Text, Text Fields, Check Boxes and a RawImage representing a mini-map on the bottom-left corner of our screen.

2. *Scripts:*
   CameraController: Attached to the camera and used to make the camera follow the player smoothly.
   CharacterControllerScript: Simple character controller script.
   AIController: Script attached on the AI objects to generate random walkable paths for the AI to follow.
   AIManager: Manages the generation and deletion of AI and collision detection. This script will play a big role on the 2nd Milestone.
   CameraFrustumGizmo, RadiusGizmo and CanvasManager: Creating gizmos for better play-testing.

2) Main objective

The main objective of this project is to not only understand the concept of efficient programming and what a big role a proper object rendering on the scenes can influence the performance of our game, but also to find a way to get used to work with the different components Unity3D offers, get a good understanding of the workflow and to apply knowledge aquired throughout our studies.

3) Development process: Train of thought, difficulties and issues

The process of creating this small demo had it's highs and lows, in this section we'll take a better look on how was my train of thought throughout the development, and end mentioning the issues and difficulties encountered on the process. A more indept guide on how the code works will come on the Scripts section.

- The first thing that i wanted to do is create the base of the project, being the character usability, the behavior of the camera and the movement of the AI, this 3 steps where my priority when i started coding.

- The camera and character behaviors where the first scripts writen, there are great tutorials on the Unity forum explaning the usage of quaternions instead of euler angles for the rotation of our main character. The creation of the CameraController script took a while, most of the interations of this script ended not being smooth and having an erroneous rotation according to our character. After reading and understanding the usage of euler angles and damp factors, i ended with a camera that follows a character in a smooth fashion.

- After i already had a movable character with a smooth camera attached to it, i started developing a simple patrolling AI. The goal was to create an AI who could freely walk to random points of our terrain, avoiding obstacles. Avoiding the other AI or the character wasn't a priority, therefore was left behind.
  The first iteration on the movement of the AI lead to undesired behaviour, it started as a simple "pick a random point and try to move there", but quickly realizing that i could save time and make it more efficient, i dove into the NavMesh world that Unity provides for pathfinding. It took a while for me to get used with all the components, but the usage of NavMesh and NavMeshAgents make it quick an easy to have an AI who walks to different destinations while avoiding obstacles on its way. While doing this i experienced a lot of issues, them most important one was the creation of the AI.
  If i wanted to create alot of AI on random positions, sometimes 2 of this game objects will be created on the same position, and by the time, since this objects contain a rigid-body and a mesh collider, the colliders kicked each colliding AI to the edges of the NavMesh, where they wouldn't be able to walk anymore. This bux was fixed on the script AIController by more efficiently creating the new random positions.

- After finally have a working AI, it was time to create some sort of script, which will managed all the AI that live on the scene so i started working on the AIManages. The AI manager contains most of the logic of our AI and i thought of it as a container for all the different components which will be use on the future to inspect and play with this components for a better visualizationg of the efficient rendering process. Some functionalities are: The creation and deletion of new AI on runtime, the management of the list of AI created, collision detection for the camera's field of view and the lightweight sphere for the activation/deactivation of the AI.

- Alot of debugging had to be made when managing the AI, having two 'enviroments' when working with GameObjects and structs created alot of bugs on the process, since our GameObjects mobility is cause by using NavMesh and NavMeshAgents. This became clear when passing the newly created target position from the struct on the AIManager to the NavMeshAgent on the AIController scripts. It took me a time to realize that passing this values will happend before the NavMeshAgent could be activated, therefore throwing NullPointExceptions.

- The methods CameraFrustumGizmo and RadiusGizmo are simple Gizmo methods to draw a wire representation of the RadiusRenderer and the field of view of our camera. They utilize OpenGL calls to draw simple lines on the shader. The script AIManager also posseses a method to draw with OpenGL the 'unseen' structs that are constantly getting updated.

- The script CanvasRenderer was the last step on this project, and it manages the passing of data from the different Gizmo scripts to the canvas. Thanks to this method the project can work as a Standalone application and doesn't require the usage of Unity's inspector for testing.

4) Scripting section

The functionality of each script will be explained on this section.

CharacterControllerScript:

This is a simple scripts that uses the Rigid Body of the character and Quaternions to move and rotate our player around the scene.

It posseses 3 important methods:

- *GetInput():* It catches the inputs of the keyboard
- *Turn():* Used to rotate our character
- *Run():* Used to move the character through the scene, this method gets called on FixedUpdate(), according to the Unity documentation FixUpdate() is ued for updating physics, usually when updating the RigidBody of a GameObject.

CameraController:

This script is attached to the camera. Used to follow the character in a 'smooth' fashion. There are 3 methods on this script:

- *MoveToTarget():* It sets our camera to always be behind of our character.
- *LookAtTarget():* Makes our camera to constatly look at the back of our target, we smooth this up using the damping functions provided by Unity.
- *LateUpdate():* This is an important method to use when updating elements that need to be updated the latest possible frame. Since we want to move when our character have moved, to avoid undesired results, we wait to the last possible frame before moving the camera. Also recommended on the Unity documentation.

AIController:

In this script we handle AI movement. On the first iteration this script used to be on controll of the character movement aswell as collision detection with obstacles, but since NavMeshes already have integrated methods that are easier to use, i scrapped most of the code out and left as little a possible.

The idea behind this script is create random walkable positions inside our terrain for our AI to walk to. Is important to know that the movement itself is handled by Unity's internal NavMesh system.

Here we can also find 3 important methods:

- *Walk():* We check whether it is possible for our AI to move, and if so, we use NavMesh's SetDestination() to move.
- *RandomPos():* Here we create a new random position and with help with the method SamplePosition() we can check, whether this new position is on walkable terrain or not.
- *SetValue():* Last but not least, this small method is indeed a very important one. It helps keep consistency on the position that our AI is walking to. Since updating the lightweight values and the values on the AI GameObject itselft tend to happend on different initialization times, this method is used to let AIController script and the AIManager script pass data between each other and always have a consistent position.

AIManager:

Here is where the main logic of our program takes places.

The basic idea behind it isn't that hard to understand: Our AI can be on 1 state out of 3 at any given time.

- It is either inside the camera's field of view, and therefore fully rendered.
- It is not inside our camera's field of view but close enough to our player to still be updated.
- It is too far away, therefore deactivated. At this point the position doesn't need to be updated anymore.

We have two lists containing our AI on this section: One on charge of storing all the GameObjects leaving on the scene, and one storing a self defined struct containing the position and direction of the AI that isn't infront of the camera anymore.

On the first iteration of this list of structs, everytime an AI will come back to the screen, we'll then remove it's reference from the list of structs to only contain the ones actually getting updated.

I tried using Csharp *Dictionary* data structure to try to map this the AI structs to their GameObject counterpart with their indexes, but i encountered alot of issues when trying to retrieve said structs.

Then i opted to try using a *List<>()* data structure to be able to easly retrieve the needed structs without having to keep track of the indexes. Utilizing the method *Find()* it made it easy to find structs, but needing to iterate through the list everytime an element got in and out of the screen, made it unefficient and time consuming, turning into a really expensive operation when managing the list.

So at last it was kept simple: Two arrays, one for the GameObjects and one for the structs. Both counterparts having the same indexes, the only difference was adding a boolean value into our struct to check, whether the struct should be activated or not.

Two methods rule this script: *manageList()* and *manageRendering()* in that order.

- *ManageList():* As the name says, used to managing the elements of the list. It allows to create and remove AI on runtime for better play-testing. Whenever more AI is needed, calculate the difference, create and start the new AI. When we want less AI that the amount already contained, remove the difference from both lists and destroy the GameObject reference.

- *ManageRendering():* Here is where the magic happens. We are constantly iterating through both lists checking, whether an AI should be drawn or not and there are 3 steps for this:

    1. Is the AI infront of the camera? Grab the existing data if its possible and pass it to the GameObject, where the NavMeshAgent can keep moving on the same path and activate the GameObject.
    2. Is the AI on the lightweight sphere? If the last frame the struct was already active, just keep updating its position, else we take the values from the GameObject, deactivate this and activate the struct.
    3. Whenever the AI is too far away to either be infront of the camera or inside the lightweight sphere, we just deactivate it till we are close enough again.

Another really import function is *updateStruct(),* responsible for generating random positions for our AI struct to move. It also utilizes RayCasting as *pseudo* obstacle avoidance technique.

This script also contains other methods that are used as helper functions for the last two methods. Other than that we also have a *OnRenderObject()* method, which is just overriding Unity's method on charge of drawing GL components into the screen.

OnRenderObject() in our context is used for play-testing purposes and the idea behind it is pretty simple. Checking, whether our structs are being properly handled and updated by our logic is easier to do with some kind of graphical feedback. With *OnRenderObject()* we did a serie of OpenGL calls to work directly with the shader an draw simple figures on the screen. I end representing the AI structs as triangles, where you can see better the direction they are going.

RadiusGizmo and CameraFrustumGizmo:

This both methods perfom the same function, and their main purpose is to handle GL calls to draw on screen.

RadiusGizmo is on charge of drawing the lightweight sphere.
CameraFrustumGizmo is on charge of drawing the Frustum or field of view of the camera.

CanvasManager:

Contains some Text Fields and Check Boxes where the inputs can be changed. It manages the activation and deactivating of the OpenGL calls through our project and allows the proyect to become a proper Standalone play-testing application.