Rendering with Flexible Cameras Parameters Based on Habitat-Matterport 3D (HM3D) Dataset

Kangwei Liao (kliao005@uottawa.ca)

Supervised by professor Jochen Lang (jlang@uottawa.ca)

Faculty of Engineering, University of Ottawa,

75 Laurier Ave. E, Ottawa, ON K1N 6N5

ABSTRACT

This paper proposes a program made with the Unity engine that can set flexible camera configurations (regular pinhole, fisheye and panoramic lenses) in a semi-realistic rendering environment (Habitat-Matterport 3D Dataset). The program utilizes a point-in-polygon (PIP) algorithm to spawn cameras at random positions in a model. Project's related files are available on Github: https://github.com/KangweiLIAO/HM3D-FlexibleCameras.

I. INTRODUCTION

As the capability of personal computers (PC) and portable devices is vastly evolving, the appliance of computer graphics plays an essential role in the daily use of computers. For example, 3D engines like Unity, a cross-platform game engine widely used in the industry, have created many remarkable applications. In the meantime, as the new revolution of artificial intelligence (AI) is coming, developing AI agents to assist our daily activities is what we are seeking nowadays. As a result, the demand for high-quality 3D indoor environment datasets captured from the real world like Habitat-Matterport 3D Dataset (HM3D) has increased these years [1].

By combining the largest-ever dataset of 3D indoor spaces (i.e., HM3D) [1] with Unity, this project implemented different camera parameters to create flexible views in the 3D spaces. This project implemented three types of cameras: a regular pinhole camera, a fisheye camera, and a *panoramic view* camera.

A. Generating Cubemap Instances with Regular Pinhole Cameras

Like in other 3D modelling or editing software, Unity provided built-in regular pinhole cameras with basic parameters, such as field of view (FOV), clipping planes and projection methods. Therefore, the first part of this project focuses on implementing cube mapping with built-in pinhole cameras. Cube mapping with a regular pinhole camera can generate a cubemap texture (see. Fig. 1.1) consisting of six rectangular textures representing six faces of a 3D cube. The most significant advantage of using cubemap texture is that its texture coordinate could be easily represented by a direction vector from the cube centre to the value accessed. Real-time processing of 3D graphics is still expensive nowadays. To create a sense of 3D spaces, cubemap texture is usually applied to a cubic skybox to create the scene background. With a camera placed in the centre of the skybox, an illusion of being surrounded by 3D objects will be made without the actual 3D models being rendered.

With the same mechanism, this project proposes a program that can generate a series of textured 3D cubes with a regular pinhole camera at the centre of each cube. The cubemap textures of the cube instances were randomly captured inside an HM3D model (i.e. 3D model of a house). By doing this, potential users (e.g., virtual visitors) can navigate within the 3D spaces (e.g., real scanned houses) through these cubes without actually loading the 3D model of the spaces.

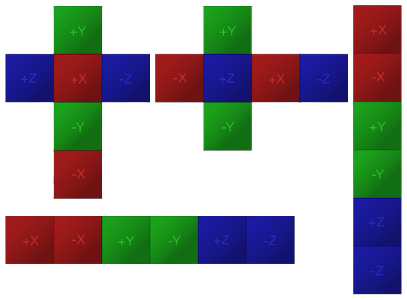

Fig. 1.1. Six faces cubemap layout [2]

B. Generating Panoramic Images with Cube Mapping

Panoramic images display a wider view than the human eyes (usually greater than 160° by 75°) with an aspect ratio greater or equal to 2:1. As a highly integrated 3D engine, Unity provided some useful cubemap parameters, such as the cubemap layouts. By modifying this parameter to a ‘Lat-Long’ (Latitude-Longitude/ cylindrical) layout (see Fig. 1.2.), a regular cubemap texture captured by a regular pinhole camera will be turned into a render texture that is often used as a panorama image. Afterwards, the program can export these panoramic render textures captured at random positions inside the HM3D model in regular image format (e.g., jpg file).

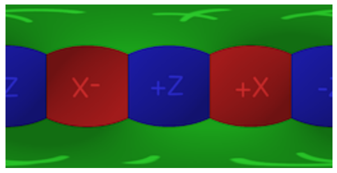

Fig. 1.2. ‘Lat-Long’ layout [2]

C. Generating Fisheye View

Fisheye lenses are used to achieve ultra-wide angles of view in photography. The angle of view of fisheye cameras varies from 100 degrees to 180 degrees. Since the image plane will become infinite when the regular pinhole camera's FOV approaches 180 degrees (see Fig. 2.), regular pinhole cameras cannot meet some special needs; for example, security cameras require the angle of view as wide as possible.

In the Unity program proposed by this project, the fisheye image is generated by five 90-degree FOV regular pinhole cameras where the corresponding directions are: top, bottom, left, right and front. Then, the program converts the five views of the cameras to five render textures and applies them to a special mesh object (see Fig. 3.) to generate the fisheye image.

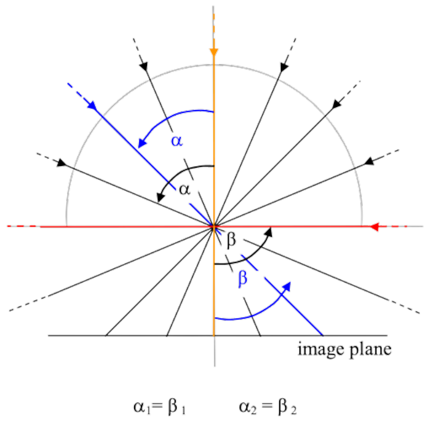

Fig. 2. Central perspective projection model [3]

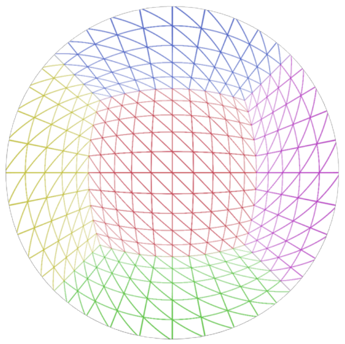

Fig. 3. The (equidistance) fisheye screen is composed of 5 sub-meshes; top (blue), bottom (green), left (yellow), right (purple) and front (red)

II. BACKGROUND

In today's game development, it has become increasingly common to use game engines to create games or AR and VR applications. Today, there are a large number of game engines available for developers to choose from, but two of the most popular game engines are currently Unity and Unreal. They are both remarkable cross-platform engines that support nearly all gaming platforms available in today's market. Therefore, which is more appropriate usually depends on personal preferences or the type of program created.

One major difference is that Unreal's source code is available on GitHub (though it is not technically open source), which means you can change any aspect of the engine yourself, including building the whole thing from scratch.

Another major difference is the graphics, Unreal is far ahead of Unity in this aspect by providing great visuals for both complex particle systems and dynamic lighting of scenes. This is why many programmers who are looking for a visual representation of their graphics will ultimately choose Unreal instead. However, no extra scene light will be added to this project since the HM3D textures already captured the lights.

In terms of programming, Unreal has two main programming languages: C++ and Blueprints. C++ has a full garbage collection system and now has a live reloading system supported by Live++, which makes C++ like a scripting language in Unreal. Blueprints is a graphics-based visual programming system. It also has the ability to be transformed into C++ code, which can reduce the performance cost of using it (but worsens the build time further).

While in the Unity engine, C# is the main programing language used. One of the advantages of using it is the build time for Unity programs is much faster than Unreal. In addition, Unity covers 18 platforms (10 for unreal); the more platforms supported, the larger the potential users will have for an application.

Compare to Graphics APIs like WebGL, Unity has a more user-friendly interface and many integrated built-in methods to use; using it will vastly accelerate the development process for the project. Also, since this project is not focusing on aesthetics, using Unity is the best choice for development.

As for the Habitat-Matterport 3D Dataset (HM3D), the biggest advantage of using it is its scale, fidelity and diversity. As introduced in its paper [1], HM3D is 'Pareto optimal' in the specific sense, which no other datasets claimed before. Therefore, choosing HM3D as the semi-realistic rendering environment used in this project is also the best choice.

III.METHODOLOGY

A. Spawning Cameras at Random Positions in Mesh

Determining whether a point is inside a mesh or not is a widespread problem in game development and computer graphics [4]. A random position spawning algorithm is required to automatically capture images with flexible cameras in an HM3D model.

A model (house) in HM3D comprises chunks of textured meshes. Unfortunately, there are no noticeable orders or naming conventions for the mesh chunks. Therefore, finding a method to organize the meshes is needed in the first step for spawning random cameras in the model. Once the program obtains an organized mesh group for an HM3D model, we can apply a PIP algorithm to generate random positions for cameras spawning.

During the organization step, the program merges all the sub-meshes in the original HM3D model. Varying on the size of the models, the performance of this operation is O(n) with n as the number of sub-meshes in the target HM3D model. On the other hand, Unity has two index buffer formats: UInt16 (16-bit mesh index buffer format that supports up to 65535 vertices in a mesh) and UInt32 (32-bit mesh index buffer format that supports up to 4 billion vertices). By simply merging all the sub-meshes for all models in the dataset (HM3D-minival), find that:

- The minimum vertex count is 842992
- The maximum vertex count is 1098316
- The average vertex count is 912062

Therefore, for all the HM3D models (in the HM3D-minival dataset), UInt32 should be used to store all the vertices for the combined mesh.

After a combined mesh is obtained from the first step, the program treats the mesh as a convex shape and applies a PIP algorithm. The PIP algorithm proposed by this project is based on a triangle point picking algorithm [5]. Since a face of a mesh in Unity is defined as a triangle, we can apply the algorithm to pick a random point on the face of a mesh. In short, the triangle point picking algorithm will generate a set of uniformly distributed points in a quadrilateral and discard the points outside the desired triangle (see Fig. 4.).

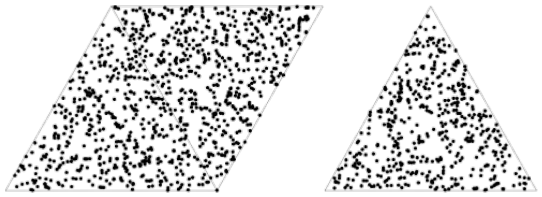

Fig. 4. Triangle point picking

In summary, the PIP algorithm executes the following steps:

1. Pick a random point A on face F1 on the combined mesh
2. Pick a random point B on face F2 on the combined mesh
3. Draw a line L connecting A and B
4. Pick an arbitrary point P on L

The program generates random points evenly spread inside the combined mesh (see Fig. 5.) using this algorithm.

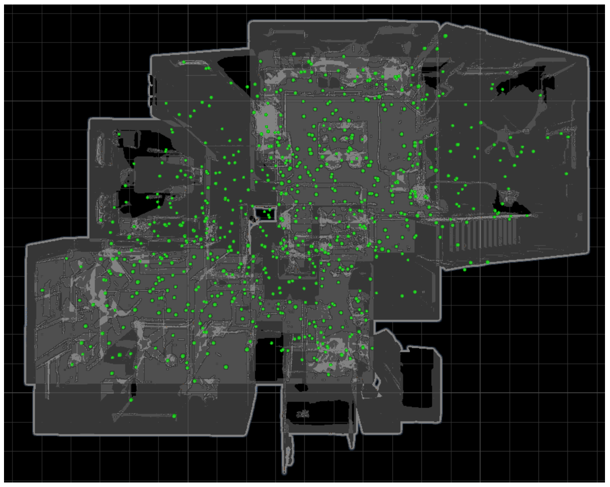

Fig. 5. Random points (green) in an HM3D model

As described in the introduction, the program will spawn a series of cube instances (navigation points). The generation of a navigation point consists of the following steps:

1. Spawn a camera C1 at a random (or selected) position P in the HM3D model.
2. Capture a cubemap texture at P using C1.
3. Destroy C1 and spawn a 3D cube with a camera C2 at the cube center.
4. Create material with the cubemap texture and apply it to the cube.

There are three types of render pipelines provided by Unity: Scriptable Render Pipeline (SRP), Universal Rendering Pipeline (URP) and High Definition Render Pipeline (HDRP). The differences between the different render pipelines are their capability and performance characteristics:

- SRP is an outdated default pipeline that has limited customization options.
- URP is an updated pipeline with quick and easy configurations to achieve optimized graphics.
- HDRP is a pipeline that can create cutting-edge, high-fidelity graphics on high-end platforms.

In this project, URP was used because of its simplicity and operability. The culling process in Unity render pipelines decide which polygons should be drawn into the view. By default, the culling method is set to ‘Cull back’, which only renders the polygons (e.g., faces of a cube) that are facing front (i.e. where its normals are pointing). Because the camera for a navigation point is placed inside the cube, we need a shader to config the culling process in the render pipeline to prevent the camera from culling the polygons inside the cube. There are three types of culling in Unity:

- Cull back, which does not render the polygons facing back (see Fig. 6.1).
- Cull front, which does not render the polygons facing front (see Fig. 6.2).
- Cull off, which will render all polygons in the view.

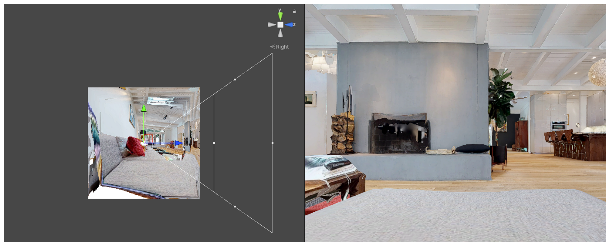

Fig. 6.1 Navigation point with ‘cull back’ view (the view is empty since the polygons inside were culled)

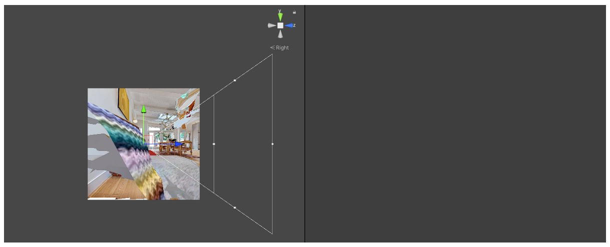

Fig. 6.2 Navigation point with ‘cull front’ view

B. Implementing Fisheye Camera

Since there is no direct way to manipulate the projection model in Unity, implementing a fisheye camera effect usually uses post-processing. However, there is another way proposed by an article [6] to create a virtual fisheye camera without using post-processing. The basic concept is that the virtual fisheye camera does not directly capture a fisheye view but captures a screen (see Fig. 3.) that shows a fisheye image. A camera rig (see Fig. 7) composed of 5 regular pinhole cameras will be used to render the screen to display. The article [6] states that the mesh objects for the fisheye screen could be created by applying the fisheye mapping functions for each vertex on a cube representing the camera rig. By changing the mapping function and create the meshes, we can obtain four types of fisheye images [7]:

- Stereographic, which maintains angles and creates an image with less distortion at the margin.
- Equidistance, which maintains angular distances.
- Equisolid angle, which maintains an equal area for each pixel [7].
- Orthographic, which will have less distortion at the center but more distortion on the margin

The program proposed by this project will instantiate a camera rig inside the HM3D model and a fisheye screen outside the model with an orthogonal camera looking at it. We can obtain the desired fisheye view from the orthogonal camera by moving the camera rig or spawning it at the designated position.

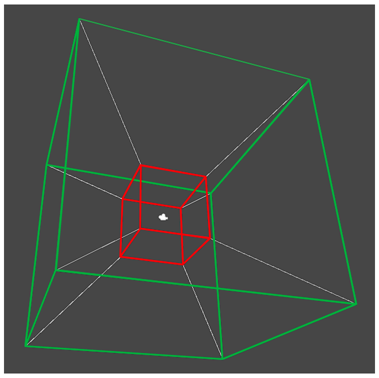

Fig. 7. Camera rig example; near planes (red), far planes (green)

IV. RESULTS AND DISCUSSION

A. Current Results

In this report, two criteria are used to classify a navigation point as qualified:

- The navigation point is not placed outside the HM3D model (i.e., the house)
- The navigation point is not stuck inside a mesh (e.g., a bed in a room or a wall between two rooms)

Based on these criteria, the average qualified navigation point for a house is 56.6% with the HM3D-minival dataset with the current random spawning algorithm. The average spawning time for 100 navigation points instances is 25 seconds; this may be different base on the IO speed of a disk since the material for a point will be stored on the disk. Also, the current program allows exporting the panoramic image utilizing a regular pinhole camera at a navigation point.

For the fisheye cameras, they can be spawned or navigate to a designated position and capture the real-time fisheye image with output to a screen.

B. Optimization approaches

The random spawning algorithm currently used for cameras is working, but it still leaves plenty of room for improvement. Most of the disqualified navigation points fall into the walls between two rooms. This is caused by simply treating the house as a convex shape. There are two approaches to improve the qualification rate:

1. Combine the meshes by rooms instead of the house in the organization step
2. Change the PIP algorithm to the even-odd algorithm

The first approach may decrease the possibility of the navigation points being placed within the walls between the rooms. However, since most of the rooms are not a convex shape, this will not bring a vast improvement.

While the second approach fundamentally solves the issue by shooting a ray from a random position in an arbitrary direction. The basic concept is that the position is inside the mesh if the number of the ray intersects with a mesh surface is an odd number; otherwise, it's outside the mesh. As proven in the paper [4], this algorithm is correct for any possible polygons. We can vastly improve the navigation points qualification rate by adopting this algorithm.

Moreover, it's redundant to use five cameras to capture the views as input for the fisheye screen. We can obtain the five camera views by capturing cubemap texture with a single camera placed at the designated position. The reduction in the number of cameras will increase the performance (i.e., reduce the spawning time of a navigation point) of the Unity program.

In addition, the instantiation of navigation point objects will halt the program due to the low performance of the built-in instantiate method; we can solve it by using object pooling for instantiation. Object pooling will create a desired number of objects when the program starts and reuse these objects instead of creating them from scratch. Also, the render pipeline of Unity will not provide dynamic batching, a draw call optimization method, for UInt32, which is the index format for the current combined mesh. Dynamic batching will reduce the number of draw calls by grouping up vertices for small meshes, which can be found plenty inside an HM3D model. Therefore, we should reduce the size of the combined mesh to less than 65535 vertices to use the UInt16 format that supports dynamic batching.

C. Faking Fisheye View

There are multiple approaches to fake a fisheye view (e.g., post-processing, some 'spherize' effects). During the development of this project, an approach of faking a fisheye view in Unity was found without using post-processing (see Fig. 8.).

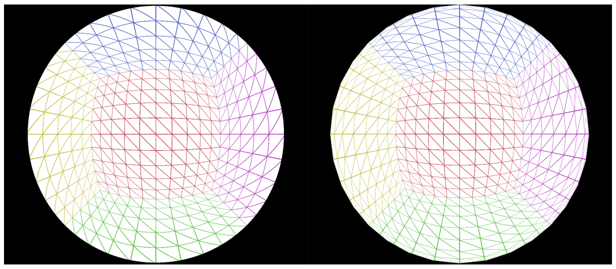

Fig. 8. Real fisheye lens (left); fake fisheye lens (right)

By moving the pinhole camera that was placed inside the navigation point outside, we can obtain a 'spherize' effect that is similar to the real fisheye image. As stated in the appendix of this article [8], by using a spherical mirror (see Fig. 9.), an approximation of a fisheye image can be created. One of the main differences between the real and fake can be found in the margin. Any of the fisheye projection functions will not create distortion shown in the fake fisheye image. The distortion is caused by the pixel being compressed at the margin of the spherical mirror.

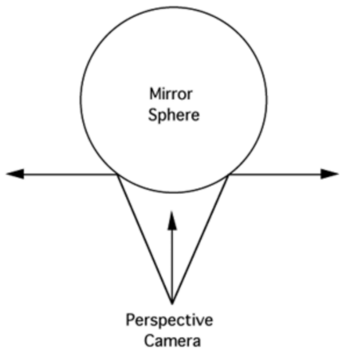

Fig. 9. Concept model of faking fisheye camera with a spherical mirror and regular pinhole camera

V. CONCLUSION AND OUTLOOK

In summary, this project uses a PIP algorithm base on the triangle random point picking algorithm proposed in this paper [5] to spawn cameras with flexible configurations. There are three types of flexible camera configurations implemented with Unity:

- Regular pinhole camera
- Panoramic camera
- Fisheye camera

The program proposed by this project achieved 56.6% of the qualification rate for the random navigation points. The performance of combining mesh is O(n) with n as the number of sub-meshes in an HM3D model. In addition, this project proposed a new method to fake the fisheye view without using post-processing in Unity.

Future works will be replacing the current PIP algorithm with the even-odd algorithm to obtain an expected qualification rate of 100% for navigation points. Also, the performance of the Unity program can be improved by reducing the number of cameras for the fisheye screen capturing, obtaining smaller combined mesh and introducing objects pooling.

REFERENCES

1. S. K. Ramakrishnan, A. Gokaslan, E. Wijmans, O. Maksymets, A. Clegg, J. M. Turner, E. Undersander, W. Galuba, A. Westbury, A. X. Chang, M. Savva, Y. Zhao, and D. Batra, “Habitat-matterport 3D dataset (HM3D): 1000 large-scale 3D...,” OpenReview, 07-Nov-2021. [Online]. Available: https://openreview.net/forum?id=-v4OuqNs5P. [Accessed: 13-Apr-2022].
2. Unity Technologies, “Cubemaps,” *Unity Manual*, 09-Apr-2022. [Online]. Available: https://docs.unity3d.com/Manual/class-Cubemap.html. [Accessed: 14-Apr-2022].
3. E. Schwalbe, “Geometric modelling and calibration of fisheye lens camera systems,” *ISPRS*, May-2012. [Online]. Available: https://www.isprs.org/proceedings/XXXVI/5-W8/Paper/PanoWS_Berlin2005_Schwalbe.pdf [Accessed: 19-Apr-2022].
4. M. Galetzka and P. O. Glauner, “A simple and correct even-odd algorithm for the point-in-polygon problem for complex polygons,” *arXiv.org*, 22-Jul-2017. [Online]. Available: https://arxiv.org/abs/1207.3502. [Accessed: 15-Apr-2022].
5. E. Weisstein, “Triangle Point Picking,” *Wolfram MathWorld*, 15-Feb-2006. [Online]. Available: https://mathworld.wolfram.com/TrianglePointPicking.html. [Accessed: 16-Apr-2022].
6. K. Park, “Fisheye Mesh Generator: How Does It Work?,” *Notion*, Jul-2020. [Online]. Available: https://www.notion.so/Fisheye-Mesh-Generator-How-Does-It-Work-e7ccf209708041e4aec295d53567cced. [Accessed: 20-Apr-2022].
7. P. Bourke, “Classification of fisheye mappings,” paulbourke.net, Jun-2017. [Online]. Available: http://paulbourke.net/dome/fisheyetypes/#:~:text=As%20the%20name%20suggests%2C%20equisolid,to%20be%20less%20in%20practice. [Accessed: 23-Apr-2022].
8. P. Bourke, “Computer generated angular fisheye projections,” *paulbourke.net*, May-2020. [Online]. Available: http://paulbourke.net/dome/fisheye/. [Accessed: 24-Apr-2022].