Shadow art

Elena Franchini

29/06/2023





Contents

1. Introduction

The goal is trying to reproduce a simplified version of the paper written by Mark Pauly and Niloy J. Mitra.
The authors of the paper have been able to create a shadow hull given some binary input images and projection information. The purpose of the hull is the ability to project different images depending on the locations of the lights and the active pixels in each input image (pixels which will form the projected shape). When more than two images are given there are often inconsistencies due to the constrains to generate the shadow hull with respect to the active pixels of each image, then an optimisation process is required to get as result the projected images that are as close as possible to the input ones.

For this purpose I used Blender associated with Python (using the package bpy), such that a user can call the Python script passing command line arguments, and blender will be opened showing the corresponding shadow hull.

2. Parameters

2.1. Images and lights positions

To simplify the project I decided to limit some parameters. The user will be able to pass from one to four binary images of size 9x9 pixels, and their position will depend on the angles passed. It is possible to specify at most four angles: 0, 90, 180 and 270 degrees:

- 0 corresponds to a light placed in the positive x axis [50,0,4]
- 90 corresponds to a light placed in the positive y axis [0,50,4]
- 180 corresponds to a light placed in the negative x axis [-50,0,4]
- 270 corresponds to a light placed in the negative y axis [0,-50,4]

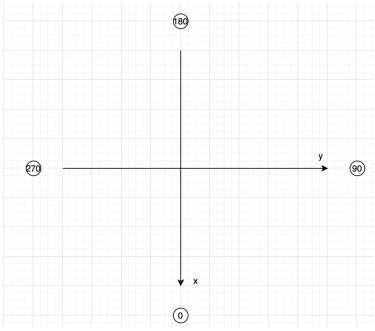


Figure 1: Lights positions

The images will be placed in front of the lights, and active pixels have values 1 (will be projected as shadow) and inactive pixels 0.

Figure 2: Images and lights configuration

2.2. Command-line arguments

To run the project is required to install Blender, but this report will be enough to understand what has been implemented. The command to execute the python script has the following structure:

```
blender --python script.py -- [nr_images] [angles] [filenames]
```

- nr_images: int value from 1 to 4
- angles: from one to four int values among 0,90,180,270
- filenames: names of the files containing the images

The number of angles and filenames should correspond to the number of images given in the first parameter, and the order of filenames should correspond to the order of angles given (i.e. if the first angle parameter is 0, the first image file passed will be created at the angle 0).
The filenames must have a .txt extension, and the image should be written as a 9x9 grid (bidimensional array) of binary values. For example, to create an image with active pixels only on the bottom boundary, we have to write in the .txt file:

```
[[1,1,1,1,1,1,1,1,1], [0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

3. Implementation

3.1. Images creation

The images have fixed size and four possible positions given by the angles. I decided to use planes which represent pixels, and each plane has a fixed size of 1 cell. Knowing the position of the lights, each image is created such that it is in front and centered with respect to the light origin.
Knowing all the centers of the planes which will compose the pixels of the image, it is possible to create them using the method bpy.ops.mesh.primitive_plane_add(). To distinguish the pixels, the material is added to each pixel and the white colour is used for inactive pixels (value of 0), while the black one for active pixels (value of 1).

Figure 3: Image with active and inactive pixels

3.2. Lights creation

Creating the lights objects is trivial since we already know the locations, then using the built-in method bpy.objects.new() we can create any light object at any location. To make the project more interesting, I used point lights instead of directional lights (because in this way the projection is more complex and realistic).

3.3. Images rays creation

At the beginning my idea was to create the voxels using the rays starting from the light origin of each image and passing through the center of each pixel, and the voxels woud have been created where the rays of all images intersect. In addition, the rays of the corner pixels would have been used to delimit the boundary of the entire hull shadow space.

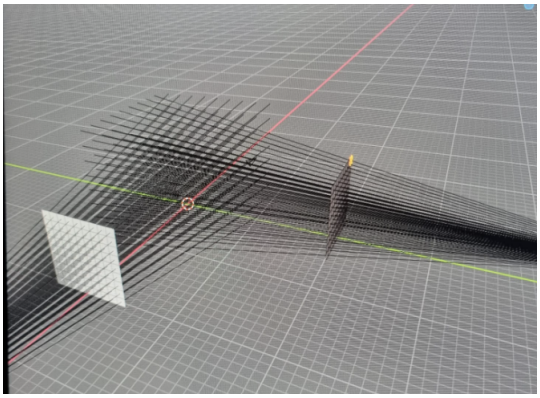


Figure 4: Pixels rays

The approach resulted inefficient and non-trivial mainly because of the lights structure: using an orthographic projection would be easier since the rays intersect perfectly and we can create the voxels at the intersection points. With a perspective projection we cannot use this approach if we want to maintain the same size of the voxels as the one of the pixels. In addition, the rays from the corner pixels don't intersect, then it was not possible to get the hull shadow space.

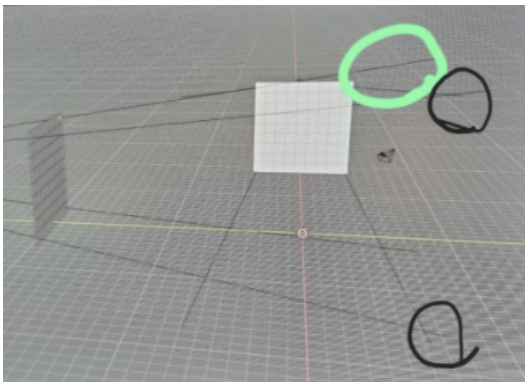


Figure 5: Hull shadow space definition with ray intersection

It is possible to see in the figure 5 that some rays intersect (in the black circle) while some others don't (in the green circle), and this is caused by the perspective projection.

This idea is then been dropped for a better solution that is explained in the next two sections.

3.4. Hull shadow space creation

Before finding the active voxels I defined the whole space on which they can be contained. Since the images and the lights have fixed size, I opted for representing the space as a cube of size 20x20x20 cells positioned at the origin. By looking at the rays which passes through the corner pixels of each image following the perspective projection, this size covers them, then all the possible voxels will be taken into consideration.

3.5. Active voxels creation

Defining the active voxels has been non-trivial. The main goal is to check for each voxel that all its rays starting from the voxel center to the origin of each light intersect an active pixel for each image. If this condition is met, the voxel is active and can be created.

At first I define the rays for each voxel to the lights origins depending on the number of input images given (i.e. if the user passes three images, each voxel will have three rays because there will be three lights). I iterate over all the existing rays for each voxel, and before checking if they intersect an active pixel I perform a standard ray-plane intersection to check if the ray actually intersect the plane on which the image lies (to drop immediately the rays we don't need).

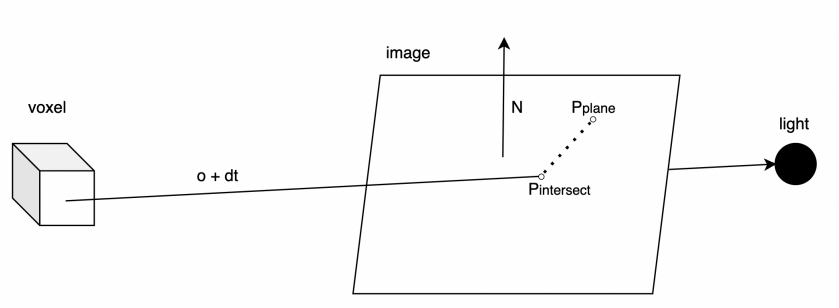


Figure 6: Ray plane intersection

Initially we check if the dot product between the normal of the plane and the voxel ray is different from 0, such that we avoid the cases in which they're parallel. In the following step, we compute t using $t = \frac{<p-o,N>}{d,N}$, and then we find the point of intersection solving $o + dt$. When the points for each image with respect to a single voxel center are computed, I have to check that these points are inside an active pixel for all images. To do that I stored the corner coordinates of each pixel center and I check that the points are inside the boundary defined by the corners. If this condition is respected, the voxel is active and it is created.

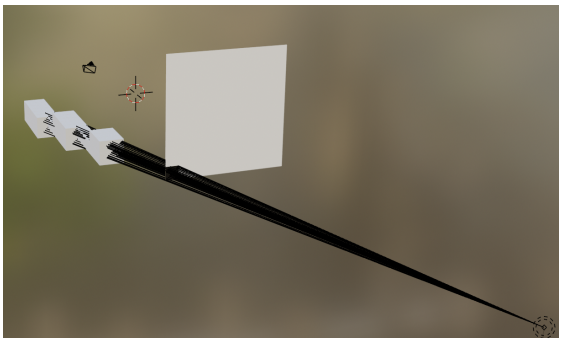


Figure 7: Voxels with one image

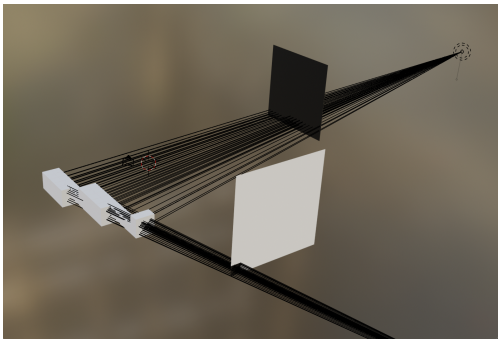


Figure 8: Voxels with two images

Another challenge was to create the voxels, since it had a huge memory requirement due to the big number of voxels. To solve this problem I had to create all the voxels inside an empty object. In this way all voxels as considered as a single object and the complexity is reduced.

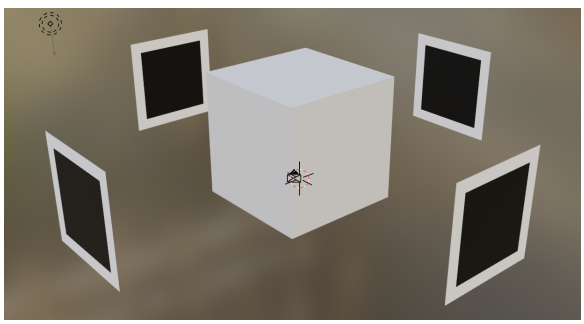
3.6. Optimisation implementation

Due to the workload of the semester I only implement a small part of the implementation. When there are more than one input image often there are inconsistencies in the projection of the shadows because of the constraints to create the voxels. To solve this problem, small modification on the pixels positions of the images which create these inconsistencies are made, such that the result is satisfiable with the minimal deformation for the shadows. If there is an active pixel in one image that is not present in its shadow projection, there is a line of empty voxels from that pixel (called inconsistent pixel) to the shadow pixel (the projected one that is not visualised due to the inconsistency). As solution one (or more) of the other images are deformed by assigning a cost to this violation and the least-cost voxel provides the position constraint for the deformation.
To search for the inconsistent pixels, I have to look for the active pixels whose voxels are not created due to the inconsistency of the active pixels of other images, then I store all active pixels that allow create voxels (but not all are actually created), and I remove the pixels that actually permit to create voxels (getting the inconsistent pixels).
In the next step I search for the line of voxels associated with inconsistent pixels, and with the perspective projection is more complex since we don't have straight lines. I implemented the list of empty voxels, but they're not associated with the correct inconsistent pixel.

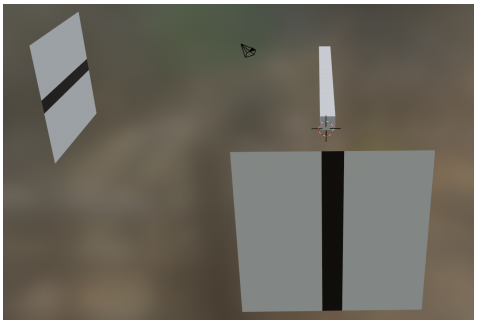
4. Results

The accuracy of the results is limited due to the fact that the optimisation part has not been implemented, but for the implemented part the results respect the directives of the paper. If we use images with very different shapes, the resulting active voxels are very

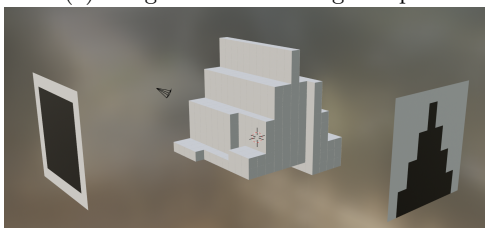
limited (again because the missing optimisation part). Using similar pixel shapes the result is similar to these shapes, and the shadow the voxels will produce will be similar to the input images.



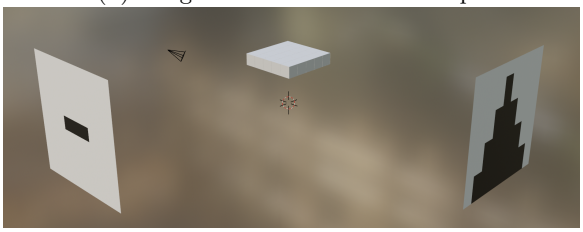
(a) Images with same large shape



(b) Images with same rotated shape



(c) Images with similar shapes size



(d) Images with different shapes size

Figure 9: Shadow hull result