ANKARA UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

BLM 4062 PROJECT REPORT

Dynamic Pixel Perfect Shadows In 2D Scenes

Alp Ertunga Elgün

19290238

Doc. Dr. Gazi Erkan Bostancı

June 2023

ABSTRACT

There are lots of diffirent illumination technuqies in 3D graphics that can produce real looking scenes with shadows. Most of those technuqies uses pre-computed light maps to bake the light data before runtime. And then use that light maps to illimunate the scene. But some more advanced technuqies do not depend on any pre-computed data. They can illimunate the scene in real time, which means they can adapt quickly if any changes made to the scene in runtime. Although these algorithms are slower than their pre-computed conterpatrs, modern devices are fast enought to run more comples algorithms.

But for 2D scenes, none of the techniques that worked in a 3D scene works. They require special illumination techniques. In this project, we studied those different 2D illumination techniques and figured out which one works best for a variety of scenes. Then we developed this technique as an example that can be run in a web browser.

You can find the source code of the project in its github repository: https://github.com/Sayuris1/Dynamic-Pixel-Perfect-Shadows-In-2D-Scenes

You need Lua and C++ to compile the source code. We suggest that you follow the instructions on this website to download and install Lua: https://www.lua.org/download.html

If you want to try out the example project, then you need a GPU and a web browser that supports WebGL 2.0.

Try out from this link:
https://sayuris1.github.io/Dynamic-Pixel-Perfect-Shadows-In-2D-Scenes/

Table of Contents

1. LITERATURE REVIEW

1.1. Scene Geometry Based Techniques

1.1.1. Screen Space Lightmaps

Lightmaps are simple textures which store incident light information about an object or location. For 3D graphics generating and using them is a huge topic, but for 2D graphics this topic is vastly simpler. [1]

One of the simplest algorithms that have been used since 1991 creates an off-screen texture that fills the whole screen, renders basic light sprites to that off-screen texture, then blends this off-screen texture with the screen to light up the scene. Dark areas become dark, and lit areas get tinted by the light [2].

In The Legend of Zelda: A Link to the Past (1991), Nintendo used used this technique. One trick they used was not to clear the lightmap to black each frame, but a dark color. This becomes the ambient light value in the scene, the amount of light things receive when they aren't directly lit by a light. In the following example, the background color of the lightmap is a dark blue. If it was black, then anything that wasn't directly lit would be completely black too. [2]

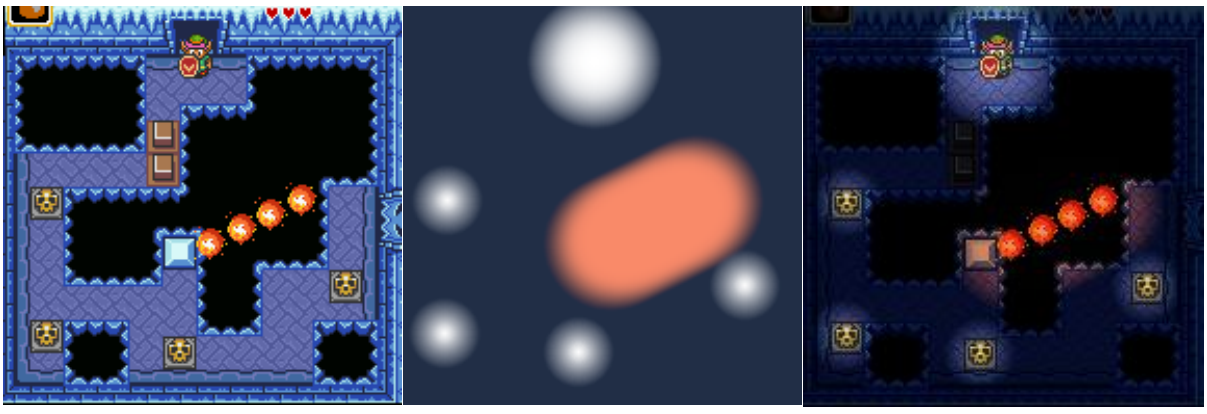

Image 1.1:
On the left: Screen without the lights.
On the middle: The lightmap.
On the right: Screen and the lightmap blended. [2]

1.1.2. Shadow Masks

To have dynamic shadows in the lightmap, lights need to light up only where the light can actually reach. Edmund Mcmillen used a simple technique in Gish, before rendering each light, he used the shadow geometry to draw a mask that blocks the light from drawing where it's not supposed to go. He found out that the easiest way to do this in most environments is to draw the shadow mask into the lightmap's alpha channel. [3]

Image: 1.2: A lightmap that has been masked with a shadow mask. [2]

In order to draw the shadow mask, he first outlined all of the objects in line segments. He was using a physics engine that only supported boxes, so he reused those collision boxes as line segments for the shadows. Then took those line segments that surrounded all of the shadow casting surfaces and projected them away from the light's origin. This turned each segment into a quad that covered the area occupied by the shadow.[3]

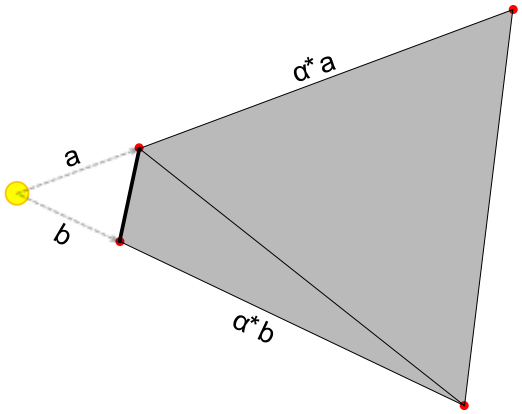

Image 1.2: Yellow dot is the light source. Red dots are the corners of the quad.

1.1.3. Visibility Polygons

Visibility region or visibility polygon is the set of points in a polygon that are visible from a reference point. Imagine standing at point holding a flashlight in a building whose walls are the polygon. The visibility region is the region illuminated by your flashlight.[4]

Visibility polygons are an alternative to drawing shadow masks. Basically, instead of masking off pixels that the light can't reach, it finds a polygon that only covers the pixels that a light can reach. This sort of flips the problem inside out.

This makes sense in software rendering because the cost to generate the visibility polygon outweighs the cost to draw the pixels needed to do shadow masking. However, with the support of hardware acceleration shadow masking wins both in terms of simplicity and performance.

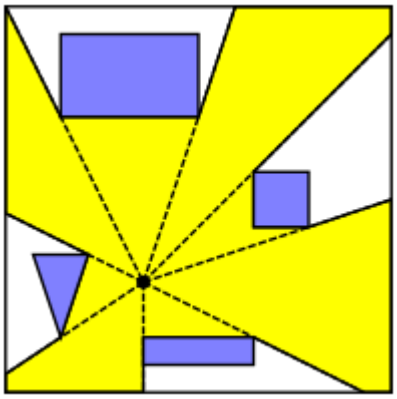

Image 1.3: Visibility polygon shown in yellow. Four obstacles are shown in blue.

1.1.4. Shoft Shadows

A point light casts only a simple shadow, called an "umbra". For a non-point or "extended" source of light, the shadow is divided into the umbra, penumbra, and antumbra. The wider the light source, the more blurred the shadow becomes. [5]

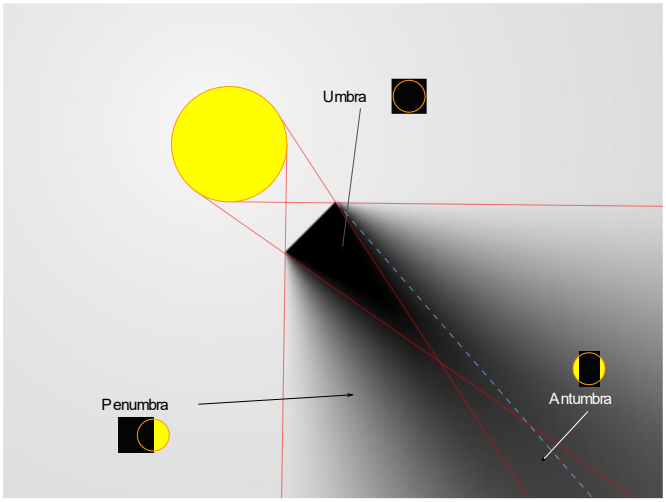

Image 1.4:
Umbra, penumbra, and antumbra with a extended light source. [2]

There is no point light in real life, so a shadow only with an umbra looks unrealistic. To make realistic shadows, an algorithm that works with extended light sources was needed. Orangy Tang solved this issue with a technique he called “Shadow Fins”. [6]

He used the shadow map technique that Edmund Mcmillen used. But instead of using the origin of the light source directly, he used an offset to simlate an extended light source. Then instead of only one quad, he created two quads. One for the $origin - offset$ and one for the $origin + offset$. [6]

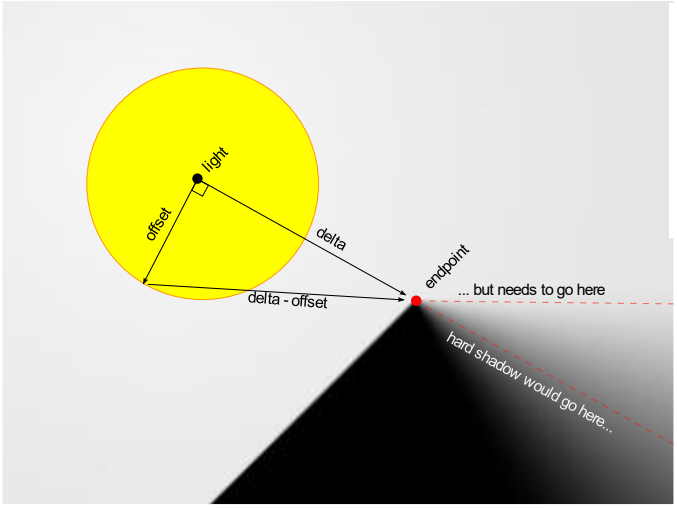

Image 1.5:
Shadow Fin creation with an offset. [2]

Then instead of directly darkening the shadow, he created a gradient with $smoothstep(-1.0, 1.0, uv.x/uv.y)$ function and used that gradient to color the shadow. Bottom half of this gradient is not used. Going from left to right and bottom to up represents going far in penumbra. Going to left and up represents going away in antumbra. [6]

To find the uv coordinates, he used $-offset$ vector as the x-axis and $delta$ vector as the y-axis. End point of the line segment were origin and end point of the quads were $(1,1)$. Then to convert from endpoint relative coordinates to penumbra gradient coordinates, all he needed to do was use the basis vectors as the columns of a matrix, and invert it. Using the adjugate function: [6]

$$mat2\ adjugate(mat2\ m)\{return\ mat2(m[1][1], -m[0][1], -m[1][0], m[0][0])$$

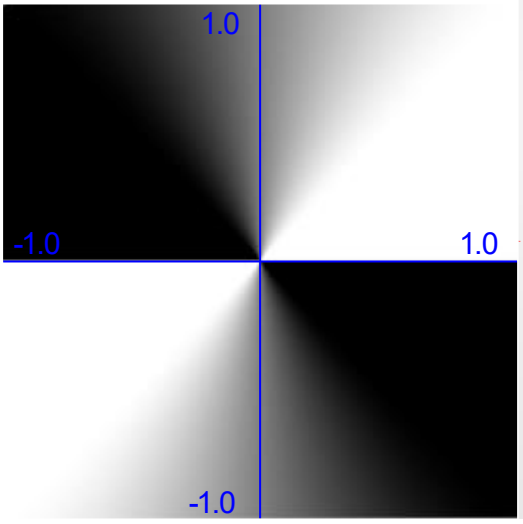

Image 1.6: Plot of smoothstep function. Bottom half is not used. [2]

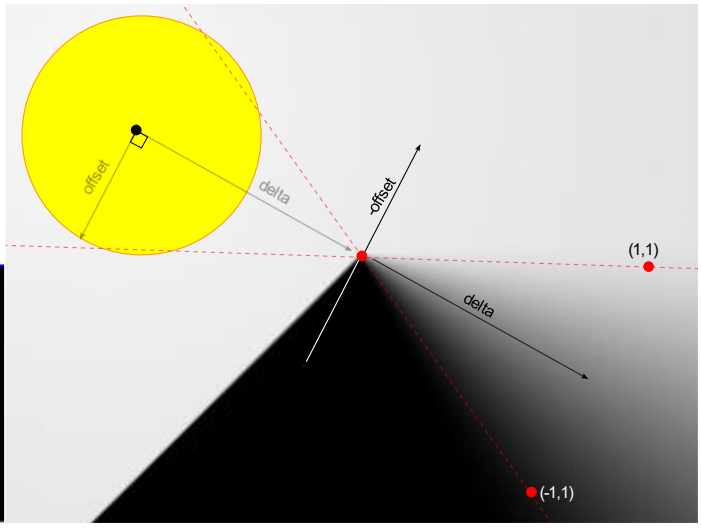

Image 1.7: Calcualting the uv coordinates. Red dots represents values in the new coordinate basis. [2]

1.2. Distance Field Based Algorithms and Global Illumination

1.2.1. Jump Flooding Algorithm

The jump flooding algorithm (JFA) is a flooding algorithm used in the construction of Voronoi diagrams and distance transforms. [7]

The jump flooding algorithm has desirable attributes in GPU computation, notably constant-time performance. However, it is only an approximate algorithm and does not always compute the correct result for every pixel, although in practice errors are few and the magnitude of errors is generally small. [7]

The distance transform calculates and stores the distance from each pixel to the closest seed, the point with zero distance. Doing that, you may get a texture like image 1.8. It's basically a pre-computed closest object map.

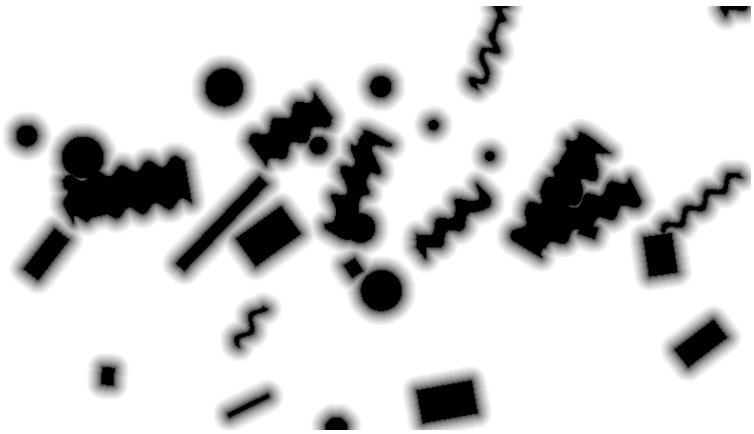

Image 1.8: A distance texture created using the jump flooding algorithm. Note that the distance is clamped at a maximum and mapped to the 0 to 1 range to make this image [8].

Jump flooding algorithm firstly prepares a $N\ x\ M$ texture the algorithm will run. Initializes the texture with some sentinel value that means “No Data” such as $(0.0, 0.0, 0.0, 0.0)$. Then renders the images that the. Each pixel that is a seed pixel needs to encode its coordinate inside of the pixel. For instance texture may store the fragment coordinates bitpacked into the color channels if the texture has 32 bit pixels, x coordinate in r,g and y coordinate in b,a.

Then for $\log2\,(\mathrm{N})$ steps where each step is a full screen pixel shader pass, in the pixel shader, jump flooding algorithm reads samples from the texture in a 3×3 pattern where the middle pixel is the current fragment. The offset between each sample on each axis is $2\text{^}(\log2(\mathrm{N})\ -\ \mathrm{passIndex}\ -\ 1)$, where $\mathrm{passIndex}$ starts at zero and ends at $\log2\,(\mathrm{N})$. This creates a Voronoi diagram that the algorithm will use to find the distance transform.[8]

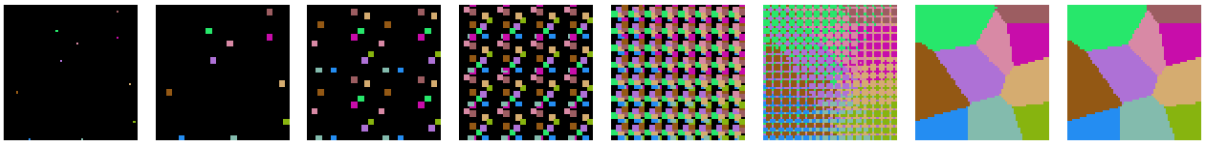

Image 1.9: A 8 pass jump flooding algorithm.

To convert the Voronoi diagram to a distance transform, the jump flooding algorithm first does another full screen shader pass where for each pixel it calculates the distance from that pixel to the seed that it stores, the closest seed location, and write the distance as output. If the algorithm runs in a normalized texture format, it has to divide it by some constant and clamp it between 0 and 1, instead of storing the raw distance value. [8]

1.2.1. Global Illumination

Global Illumination (GI) is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light). Modelling indirect lighting allows for effects that make the virtual world seem more realistic and connected, since objects affect each other's appearance. One classic example is 'color bleeding' where, for example, sunlight hitting a red sofa will cause the red light to be bounced onto the wall behind it. Another is when sunlight hits the floor at the opening of a cave and bounces around inside so the inner parts of the cave are illuminated too. [11]

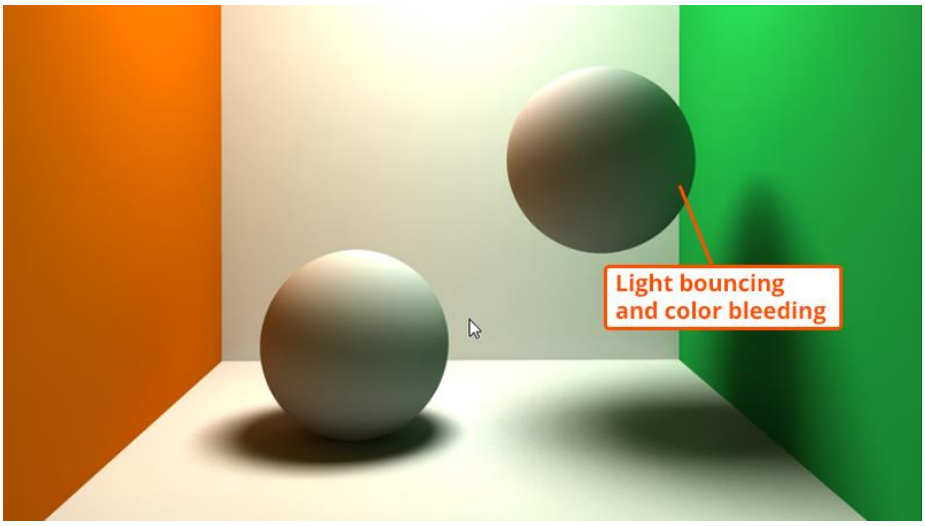

Image 1.10: Color bleeding. [11]

One solution to the global illumination problem is using ray tracing. This means following rays from a light source to the spectating eye as the ray reflects of different materials. [11] But casting rays i costly and if we want to reflect the rays from different materials, this cost just grows. So computer graphics researchers needed to find a way to produce realistic looking images with low amount of rays.

At the core of most global illumination algorithms is random sampling of data to build up an increasingly accurate representation of the the ‘true’ solution as more samples are made. This is called the Monte Carlo method, and it relies on the natural tendency for randomness to converge on the correct answer given enough samples. In global illimunation, the correct answer the algorithms converging upon is the solution to the rendering equation.[10]

Johan Forslund and Pontus Arnesson used the Monte Carlo method to make a fast ray tracer. Using the Monte Carlo method, a point in the scene can send a random number of rays to random directions and with time the result will look realistic. [12]

Then for the infinite bounces, Johan Forslund and Pontus Arnesson used the Monte Carlo method quite cleverly. Basically, they saved the emission from the last frame into a buffer which gave them the direct lighting data. On the next frame they sampled the buffer when the ray has hit a surface and added that emission amount into the hit surface's emission. This gave them the data from the ray reflections.

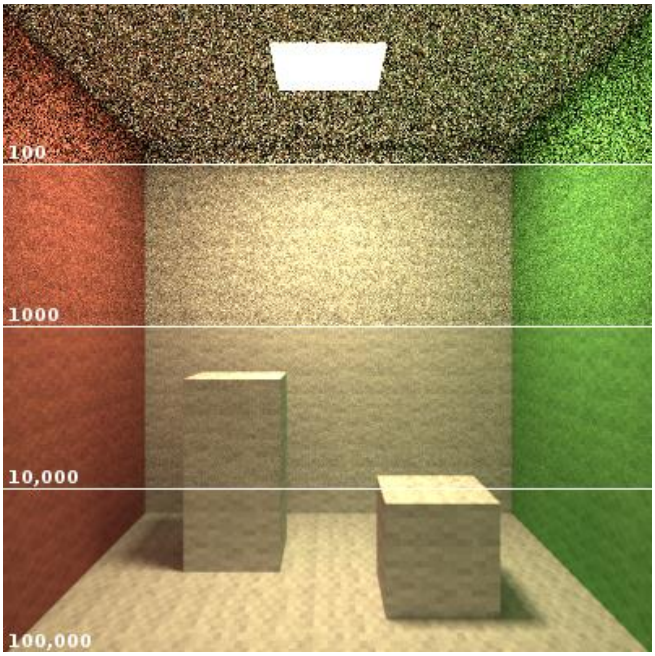

Image 1.11: Numbers on the left represents how many frames passed since the start.
With the Monte Carlo method, the imge looks more realistic and less noisy with time. [12]

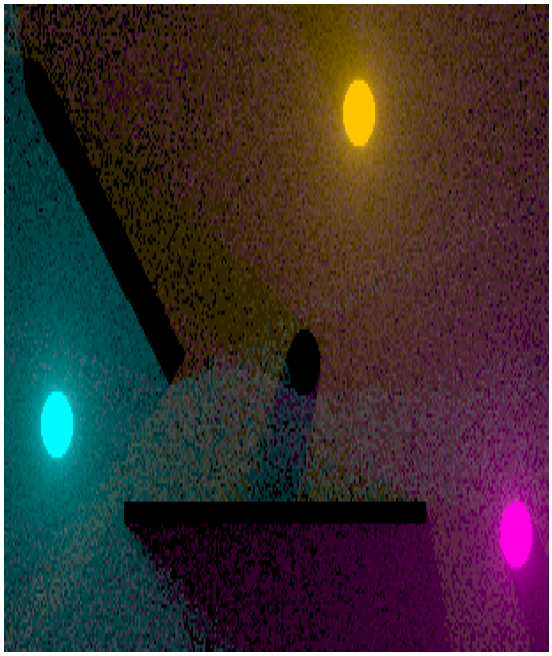

Image 1.12: Global illimunation in a 2D scene. [12]

1.3. Tile Based Algorithms

In a tile based scene, the scene consists of small square (or, much less often, rectangle, parallelogram, or hexagonal) structural units referred to as tiles laid out in a grid. Using this grid, scenes can be constructed from small images or image fragments multiple times. This results in performance and memory usage gains, big image files containing entire scenes are not needed.

Using a grid makes it easier to store other data related to the scene too. Each small tile can hold information about collision, a path-finding, neighboring tiles, and most importantly lighting. Each tile can have values like current light color, diffuse color, specularity, reflectivity. This makes lighting the scene very easy.

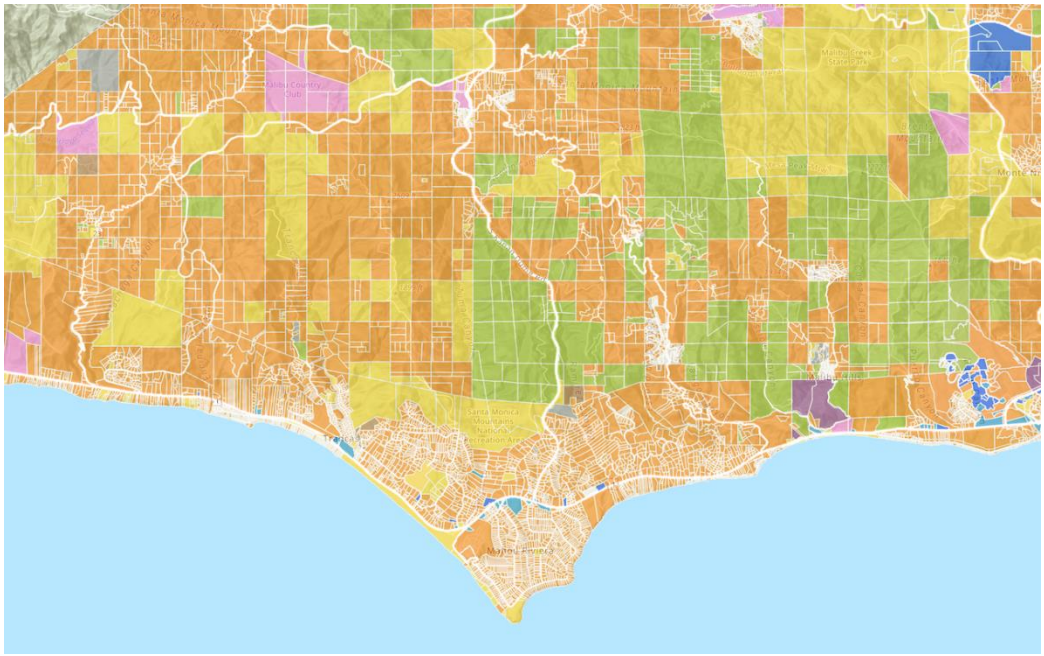

Image: 1.13: A tile map of New York. Holds height information.

1.3.1. Lights in a Tile Based Scene

The main goal for the lighting system in a tile based scene is to figure out where all the light sources are and how much intensity diffuses to each block. In most of the different illumination models, diffusion formulas use distance to light, light intensity and properties of the material that is going to lit up. [13]

Those material properties can be easily stored in each tile of the grid. But to find what lights effect that grid, what are their intensity and distance to the one tile with tile light Jeff Martin needed to use a fast algorithm. [13]

He found out the simplest way to find the distance to the light was using the Breadth-First Search algorithm. First he stored the tiles in a tree data structure instead of a simple array. This made running the Breadth-First Search algorithm easier and faster. After that he just lit up the tiles using the $LightIntensity - distanceToLight$^2 formula. [13]

The result was blocky. He calculated the distance to light only per tile, but every pixel inside those tiles had diffirent distance to light value. He solved this problem easly using the GPU to calculate the distance. GPU's have special units that can interpolate 2 values quickly. So each pixel can have diffirent distance to light source without consuming much processing power.

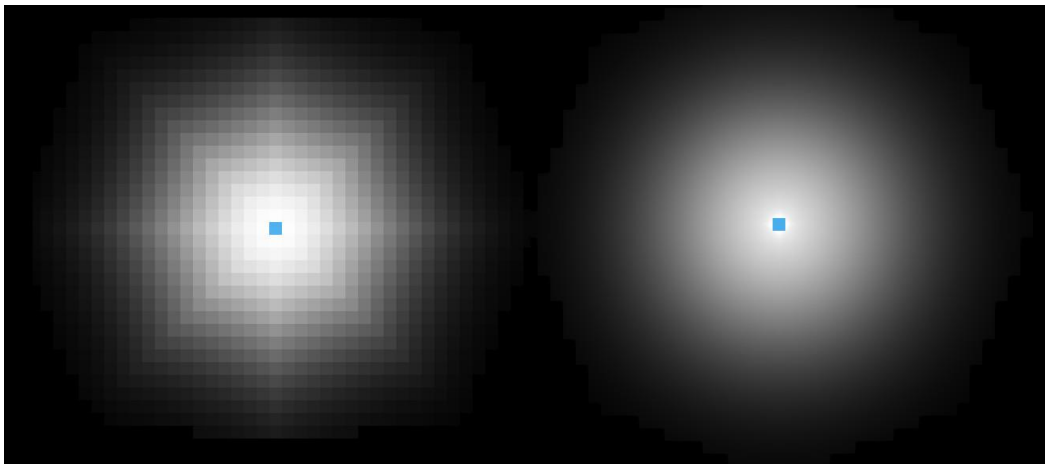

Image 1.14: On the left, result of BFS without interpolation.
On the right, result of BFS with interpolation. [13]

2. DISCUSSION

2.1. Downsides of Those Techniques

2.1.1. Downsides of the Distance Field Based Technique

All of the techniques we discussed above, with one exception, produce fast and accurate results both for lighting and shadows. That one exception is the distance field based approach.

Even though it is the only technique that can produce globally illuminated scenes, the way it works is not optimized via hardware acceleration. GPU's are optimized to work with meshes and triangles not with distance values.

Some modern GPU's optimised to run ray tracing algorithms, not with ray marching. Ray tracing is the process of taking a ray, and performing intersection tests with discrete geometric primitives, or meshes, known to be in the scene.

To be more concise, the hardware only implements very fast ray-triangle and ray-aabb intersections, and all the rest is implemented in software. The software has been optimized very precisely for each GPU's hardware though, we can rely on these implementations to be basically as optimal as we can get.
Even though we can use those optimizations for ray tracing. We can't use them for ray marching.

2.1.2. Other Techniques

Even though all of the other techniques run fast in the modern hardware. Each have some unique downsides.

If we use the shadow maps, or the light maps, like we discussed above, the performance will drop significantly with the each light we add. This makes it hard for designers to construct a scene. There will be a max number of lights they can use so they can't make the scene however they like. This would slow down their design speed and makes it hard for them to use their imagination.

And because the shadow map algorithm uses line segments to draw the shadows, designers need to outline all of the geometry they draw with line segments. This will slow down the production quite a bit. And there is no way to produce pixel perfect shadows. Every shadow will have some kind of hard line segment structure to them.

Not only that, because shadow map algorithm draw shadows and lights separately, there could be some errors to them. Finding and solving those errors would be hard since those two algorithms are compleatly independent.

And if we would like to modify the scenes in the runtime, we would need to find the new start and end points of the line segments. This would increase the complexity of the program. Which would decrese the performance and robustness of the program.

Tile base approaches solves the issues that we disscused above. But in a tile based scene, everyting need to be small squares. There won't be any way for the designers to add diffirent small shapes to the scene.

Some algorithms solved this issue by decreasing the tile size to nearly a pixel size. This way even thoght the scene consisted of little squares, they could draw diffirent shapes the scene.

But they were using a offline renderer. Considering the resolution of the modern screens, tile based algorithms can't run fast enough to use a real time renderer.

2.2. What Do We Need

We need to use an illumination algorithm that can run fast enough in modern hardware. So we can use this algorithm in real time programs like similators or games.

The algorithm that we are going to use should be able to work without additional data, like a line segment, and it shouldn't requare us to construct the scene in a spesifict way, like a tile grid.

Instead of projecting shadow from a predefined structure, like a line segment or a tile. It should be able to cast shadows for every pixel we wanted. So the shadows won't be in the shape of a little line segments, or little squares. They would be perfectly the same with the pixel on the scene. We call those shadows "pixel perfect shadows". So the algorithm should be able to produce pixel perfect shadows.

It would we better if the algorithm could support global illimunation. But in a 2D scene, there won't be big diffirances. So it is not a must.

And it would be better if the we can add unlimited light sources without effecting the performance. But those kind of algorithms are quite slow. Even thoght they would be faster if we continue to add more lights, there is no reason to use them if they are not performant enaugft to run on a modern hardware with only one light. So there is no need for the algorithm to work with unlimited number of lights.

2.3. Pixel Perfect Shadows

After some reserch, we found about Catalin Zima-Zegreanu and his pixel perfect shadows technique. His technique didn't requare any additional data and fast enough to run in modern devices. [14]

2.3.1. Preparing the Occlusion Map

The first step for each light is to render an "occlusion map" to a texture. This way, the algorithm can sample the scene and determine whether an pxiel is a shadow-caster, or not a shadow-caster, transparent. The algorithm expects the light to be at the center of this occlusion map. The algorithm uses a square power-of-two size for simplicity's sake; this will be the size of the lights falloff.. The larger the size, the greater the falloff, but also the more fill-limited the algorithm will become. [14]

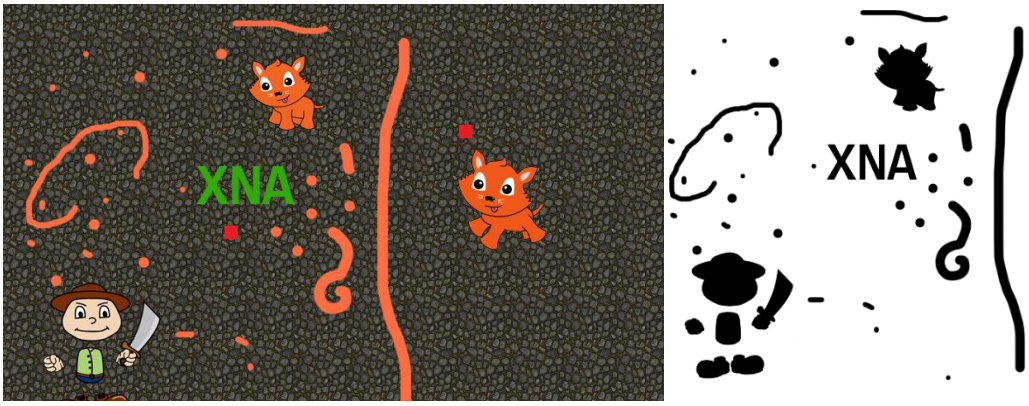

Image 2.1: The scene on the left. The occlusion map on the right. [14]

Drawing the occlusion map is a lot like drawing a shadow map in 3D scenes. In 3D scenes, each light has a depth map like the occlusion map, but instead of only storing shadow-caster information, it stores the distance from the light to that pixel. Which is called "the depth" of the pixel. Then if the actual depth of the pixel, distance from camera to a pixel, is bigger then that light source depth value, this means that pixel is in shadow. So it calculates the lighting data accordingly. [14]

2.3.2. Calculating the Depth Values

Just like a 3D scenes depth values can be stored in a 2D texture, a 2D scenes depth value can be stored in a 1D texture. To find the depth values, Catalin Zima-Zegreanu used a diffirent coordinate system, Polar coordinate system.

In mathematics, the polar coordinate system is a two-dimensional coordinate system in which each point on a plane is determined by a distance from a reference point and an angle from a reference direction. The reference point (origin of a Cartesian coordinate system) is called the pole, and the ray from the pole in the reference direction is the polar axis. The distance from the pole is called the radius, and the angle is called the angular coordinate. [15]

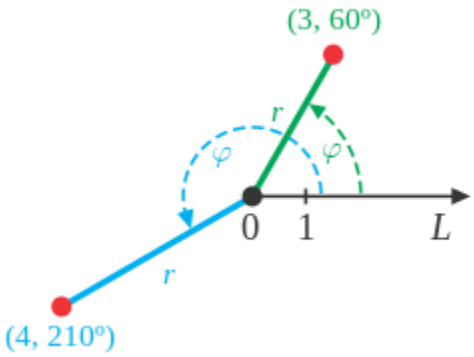

Image 2.2: Points in the polar coordinate system with pole O and polar axis L. In green, the point with radial coordinate 3 and angular coordinate 60 degrees or $(3, 60°)$. In blue, the point $(4, 210°)$. [15]

He used the polar coordinate system to transform texture coordinates of the occlusion map to a polar coordinate system. Which he used to send “rays” from the middle to the the all sides. If the rays hit a occluder, a dark pixel. He stopped and wrote the distance value to a 1D “depth map”. Indexes of that depth map was the same as the degree of the ray he sent. And the value is the distance that he got from that ray. [14]

Image 2.3: Red rays going form the light position to each side. They stop when they hit a occluder. [14]

2.3.3. Shoft Shadow and Rendering Lights

The next step he does is to render our lights by sampling from the 1D shadow map lookup texture.

The basic idea is: for each fragment in the light area, determine if the distance from center is less than the distance for that angle in our 1D shadow map. If true, the fragment is "in light." If false, the fragment is "in shadow." [14]

The resulting scene hadl lights an shadows. But there was only lights shadows with hard edges. And something was off with the lights. [14]

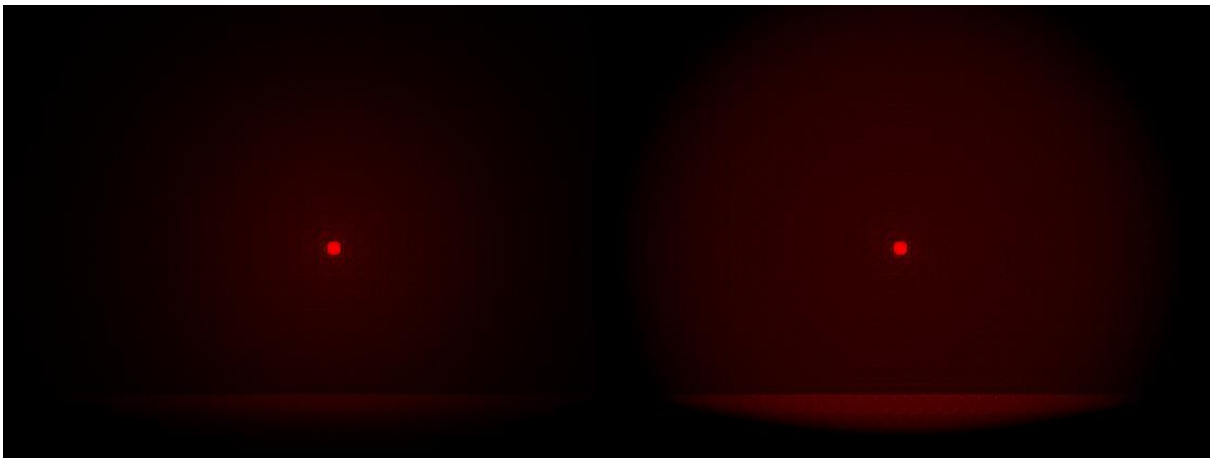

Image: 2.4: Two light sources. Both red, but they're different somehow. [13]

The one on the left looks pretty good, but the one on the right looks weird. It's hard to say why it's wrong though. It looks too bright somehow. Tums out, the attenuation of the light was wrong. Physically, light intensity falls off exponentially from the source and anything else just doesn't look right. Lighting systems are full of little details like this, and if the algorithms doesn’t get them just right, things just look weird. [14]

After solving that issue, he tried to soft shadows to the algortihm. First he tried the percentage-closer filtering method which is used in 3D shadiw maps. Unlike normal textures, shadow map textures cannot be prefiltered to remove aliasing. Instead, multiple shadow map comparisons are made per pixel and averaged together. This technique is called percentage-closer filtering (PCF) because it calculates the percentage of the surface that is closer to the light and, therefore, not in shadow. [16]

The original percentage-closer filtering algorithm, worked with mapping the region to be shaded into shadow map space and sampling that region stochastically (that is, randomly). [16]

But later he found out that just bluring the shadow map while he was rendering the shadows looked much better. And unlike percentage-closer filtering, he could apply the blur while sampling from the shadow texture. This would speed up the shoft shadow step of the algorithm significantly . [14]

The result is exactly like what we wanted. Using this technique we can get some very pretty per-pixel soft shadows in real time. Exactly how we wanted the algorithm the look like.

Image 2.5: How the scene look after pixel perfect shadows technique. With a few different lights and some additive blending, we can get some very pretty per-pixel soft shadows.

3. IMPLEMENTATION

3.1. Libraries

The main goal of this project is to implement a pixel perfect shadow technique that can be used easly by designers. And it needs to run fast in modern devices. There is no need to reinvent the wheel, we don't need to code our own renderer, window manager, input manager and resource manager. We can use a exiting high level API that handes those managers for us. But most of those API's do not let the user change how the renderer works. We need be able to modify the renderer of this high level API to draw the shdows however we want.

After some research we found out about Defold. It handels all of the things we didn't want to do all over again form the begining and most importantly, it lets us modify every aspect of the API hoverwer we want. It's written in C++ natively but has scripting capibilities with Lua. [17] Lua is a fast, dynamicly typed scpirting language that can be easly integrated to any program written in C or C++. It's syntax is simple. [18]

Because we wanted to try all of the thecniques we discussed above, we decited to use Lua. It is a very basic scpirting language, implementing ant trying new things is realy easy. So there is need to feel bad if a thecnique we tried didn't produce the results we wanted. And thakns to Defold, we didn't spend any time implementing the thinks that is not related to shadow thecniques directly.

And lastly, Defold can make solid web builds. So any reader that wants to try the pixel perfect shadow thecnique can easly try it in a web browser. [17]

3.2. Render Pipline

Before everyting else, we modifed to render pipline so it would work with the shadows. For each light, we first set the rendering origin to the light origin then rendered the occuluders. Defold has a realy easy to use function to set the rendering position. $render.set\_view$ function. It takes a matrix with the size of 4x4. Than calculates the Model-View-Projection matrix for us. We used $matrix4\_look\_at$ function to find that matrix. It takes a position, a look at position, and a up vector. Position s the positon of the light. It needs to look at down direction. And up vector is direct opposite of down direction.

Then we added some basic functions tto change the light position, size, color and angle. And some more basic functions to add or remove the lights.

Then in the shader, to transform the rectangular Cartesian coordinates to polar coordinates, we use the $(X,Y) = (-R * sin(theta), R * cos(theta))$ formula [15]. Theta is the degree of the ray, so it is the same as the u value of the occluder map. R is the size of the ray at the current step. So at the start it is equal to 0. Then for each step, we add one to it.

But u value of the map started form 0 and ended at 1. Ray degrees started at 0 radian end ended at 2 radians. So we first scaled and transfored the u value to -1 to 1 range using $u * 2.0 - 1.0$ formula. Then scaled it to the 2 radians range with $PI * 1.5 + scaled\_u * PI$ formula.

And at the end, we sampled our 1D shadow map and draw the lights. We needed to transform the polar coordinates back to the rectangular Cartesian coordinates. For that we used the $theta = atan(Y,X)$ formula [15]. Because our shadow map is 1D, there was no need for the R value.

References

1. Carnegie Mellon University: https://15466.courses.cs.cmu.edu/

2. Slembcke's Computational Corner: https://slembcke.github.io/2D-Lighting-Overview

3. Dietrich, S., 2001. Shadow techniques. In Game Developer Conference 2001 Proceedings. https://developer.download.nvidia.com/assets/gamedev/docs/GDC2K1_Shadows.pdf

4. D.-T. Lee, "Proximity and Reachability in the Plane," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Illinois, ProQuest Dissertations Publishing, 1978 7913526.

5. Wikipedia: https://en.wikipedia.org/wiki/Shadow

6. Orangy Tang: https://archive.gamedev.net/archive/reference/articles/article2032.html

7. Rong, Guodong; Tan, Tiow-Seng (2006-03-14). "Jump flooding in GPU with applications to Voronoi diagram and distance transform" (PDF). Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games. I3D '06. Redwood City, California: Association for Computing Machinery: 109–116. doi:10.1145/1111411.1111431. ISBN 978-1-59593-295-2. S2CID 7282879.

8. Rong, Guodong & Tan, Tiow-Seng. (2007). Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. Proceedings - ISVD 2007 The 4th International Symposium on Voronoi Diagrams in Science and Engineering 2007. 176 - 181. 10.1109/ISVD.2007.41.

9. Rong, Guodong and Tiow Seng Tan. “Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams.” 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007) (2007): 176-181.

10. Samuel Bigos: https://samuelbigos.github.io/posts/2dgi1-2d-global-illumination-in-godot.html

11. Ritschel, T., Dachsbacher, C., Grosch, T. and Kautz, J. (2012), The State of the Art in Interactive Global Illumination. Computer Graphics Forum, 31: 160-188. https://doi.org/10.1111/j.1467-8659.2012.02093.x

12. Johan Forslund Pontus Arnesson (2019) Monte Carlo Ray Tracer, Global_Illumination_compressed. https://assets.ctfassets.net/vyd75osjys97/5Vp4WpX9tux4HhGrjjTBVt/dff3035ab91f81e915eff912b4aac76d/Global_Illumination_compressed

13. Jeff Martin: https://www.cuchazinteractive.com/blog/nep-lights-p2-diffuse-light

14. Catalin Zima-Zegreanu: http://www.catalinzima.com/2010/07/my-technique-for-the-shader-based-dynamic-2d-shadows/

15. Brown, Richard G. (1997). Andrew M. Gleason (ed.). Advanced Mathematics: Precalculus with Discrete Mathematics and Data Analysis. Evanston, Illinois: McDougal Littell. ISBN 0-395-77114-5.

16. Reeves, W. T., D. H. Salesin, and P. L. Cook. 1987. "Rendering Antialiased Shadows with Depth Maps." Computer Graphics 21(4) (Proceedings of SIGGRAPH 87).

17. Defold: https://defold.com/

18. Lua: https://www.lua.org/