Neural augmentation for voxel raytracing

Cedric Goujon Maxime ?

ESGI ESGI

Paris, France Paris, France

[Cedric.goujon78@gmail.com](mailto:Cedric.goujon78@gmail.com) ?

Abstract

One of the major issues to voxel raytracing is the fact that we can see the discretization of space if the resolution of the voxel grid isn’t small enough. Other methods for voxel rendering are based on the idea of turning the voxel scene into polygons, requiring to regenerate a voxel model every time it is modified. Thus this paper proposes three techniques to improve voxel raytracing rendering, by estimating the normal, the position that would have been hit if it was a smooth surface, and a method for upscaling the voxel scene. The three methods proposed use the indirect of encoding of HyperNEAT, taking advantage of the weight sharing to emphasize the generalization of the trained network, facilitating the learning. It also gives the possibility for the neural network to scale to adapt to the level of precision desired.

Keywords

HyperNEAT, neural network, voxel, raytracing, neuroevolution

Introduction

Voxels are a representation space by discretizing it, making it easier to travel through this space as it has been partitioned into regions or voxels. Without this partitioning of space, unless some sort of intersection culling is performed, each ray would have to test its intersection with all the objects in the scene, a very expensive proposition. These regions of this partitioned space can be used to contain different kind of things, but in this paper will only look at voxels that are filled with somethings like colors or materials. This can kind of voxels are mostly used when objects should not be hollowed, like in the medical field, in 3D sculpting, in video games for destructible elements and procedural generation, in the rendering of fluids with techniques like metaballs. While it is not possible to rotate the voxels inside the grid, it’s possible to rotate the grid containing the voxels, the game Teardown makes demonstration of this. In Teardown each object is has its own voxel grid, allowing for the voxels to rotate in relation to the other objects of the world. But one of the major problems of rendering voxels filled by a color with raytracing is that if the resolution of the grid isn’t high enough, we can see the blocks forming our voxel scene. And even if we can’t see blocks forming the scene a cube can only have six normals making the lighting less precise than polygons that can have an infinite number of normal. Thus this paper will present a deep learning technique to estimate what the normal of the surface hit would be if the surface was smooth. This paper will also present a technique to estimate the depth map if the scene was made of smooth surfaces, and a technique to upscale a voxel scene that could potentially be used in real-time. All those techniques will be based on the indirect encoding of hyperNEAT.

HyperNEAT is a neuroevolution algorithm that is uses a compositional pattern producing networks (CPPN) to generate a neural network. In hyperNEAT all the nodes of the phenotype needs to be have an encoding representing them, we then give the encoding of two nodes to CPPN to determine whether or not they should be connected, and what their weight should be. A common type of encoding is geometrical space, but it is also possible to use other forms of encoding including for example time, lighting, acidity. Some of the advantages of this is indirect encoding its ability to create neural network that are regular, modular, and its ability to scale at different resolution. Researches have shown that it is possible to start by training the neural network on small resolution at the start, to train faster, and to finish the training at a higher resolution to polish the precision. Researches have also shown that adding a bias computed from this encoding such as the Euclidean distance between the two nodes can greatly accelerate the learning speed of the neural network. Although hyperNEAT was initially created for neuroevolution it is also possible to train through backpropagation.

In our case we benefit from the regularity of hyperNEAT, as it will make it easier to produce a neural network that generalizes well. And we also benefit from the scalability, allowing us to train the neural network once for the different level of precision that we want.

Normal estimation

The normal is estimated using 3 generic phenotype neural networks that takes a 3d kernel as an input and an output node at the position hit by the ray, the center of the 3d input kernel is the position of the voxel that was hit. Three phenotype neural networks are generated with one CPPN, each of the phenotypes corresponds to a 3d axis. They compute the value of the normal for their respecting axis. The encoding of the nodes is the relative position to the position of output node on the axis that we are currently computing. As the encoding of the input node is computed based on the encoding of the output, it is not necessary to give the position of the output node to the CPPN. A constant bias input of one is added to the CPPN inputs. The value of an input is either 0 or 1 for respectively an empty voxel or filled voxel. To summarize the CPPN is given a single variable input to compute the weight of a connection, three phenotypes are generated with this is unique CPPN, as the same rules should apply to each axis, to generate the three components of our normal vector. Alternatively to avoid generating a neural network each a ray hits a voxel, we propose to take the center of the last empty voxel the ray traveled through as the position of the output node. Making a total of eighteen (6 position x 3 axis) possible neural networks, or you can have one neural network by rotating value depending on the theoretical kernel center.

Upscaling

The upscaling method is very similar to the normal estimation method, the difference here is that position of the output node is the center of new voxel that we to create or the position for which we want to estimate if a voxel would be there if the resolution of our grid. The center of the 3d kernel is the center of the voxel that contains our new voxel. And this time we give to our CPPN the Euclidean distance between the input and output node. This time only one phenotype is needed, as we just need to know if the output value of our phenotype neural network is equal or above one.

Once we know if a voxel would be in this position, we need to determine what the color of that voxel should be. The position of the nodes of the phenotype network remains the same, and we still give same value to the CPPN. The phenotype network is used as many times as there is channel in the color type used. It takes in input the value of the channel being computed for each input voxel, the output is then divided by the number of voxels non-empty inside the kernel.

When the camera is very close to the camera even with normal estimation the users will be able to see the voxels. Upscaling the part of the voxel grid very close to the camera where the rays passes through could prevent or reduce this problem.

Depth map estimation

To decide whether or not a point of a surface is hit by a light, we need to trace a ray from the position that was hit by our initial ray, to the position of the light. If there was no collision then we know the light then we know that this surface is lit. It may not something important but it’s one more step towards higher fidelity graphics.

The first step here is to generate the depth map before generating the final image. The depth map should contain the exact values of the distance traveled by the rays, and not the classic modified depth map that you would get with OpenGL for example. This time we are using the a 2D kernel as an input of our phenotype neural network, the center of the kernel and the output node would have the position of the pixel that we want estimate the actual value. The inputs fed into the phenotype network are zero if the value of the pixel is too far from the value of the pixel of the center. Then once the output is computed we divide it by the number of pixel from which we have taken the real value. The CPPN is once again fed the Euclidean distance between the input node and the output node. Once we have estimated the how far each ray as traveled we can compute the lighting more accurately.

Conclusion and future works

TBD