Oregon State University
College of Engineering

CS 461 - Fall Term
Capstone Technology Review

November 21, 2017

3D Virtual Reality Painting

Prepared for

Dr. Kirsten Winters

Prepared by

Braxton Cuneo
of
Group 66
PolyVox

Abstract

As the member in charge of managing the back end of the PolyVox system, it is important to make decisions that ensure optimal performance and flexibility in operation. To these ends, the data representing the environment of the system will be structured as voxels in an sparse voxel octree. The tools manipulating this data will be encoded in shader code which is dynamically loaded by the system and injected into a standard tool template. Once compiled as part of this template, a tool's code is then run on the GPU managing the environment data of the system whenever the user invokes the use of that tool.

CONTENTS

1 Introduction

The PolyVox Team (Team 66), is expected to create a three-dimensional virtual reality environment where, through the use of a headset and motion control devices, a user may walk around and view the environment as well as manipulate this environment in a manner akin to painting or sculpting. Specifically, it is expected that a user may add or remove geometry from the scene as well as be able to modify the shape and material properties of geometry in the environment via the manipulation of their motion control devices.

My role in this project is to design and implement the back end of this environment, the back end being the software which performs both the rendering of the environment as well as the manipulation of the data encoding the state of the environment.

For the sake of clarity: It is assumed that the majority of this code will be executed on a graphics processing unit (GPU), although high-level management of this execution will be occurring CPU-side. This is because rendering and manipulating complex environments twice at 90 frames per second is impractical on any consumer hardware other than a GPU.

Three important questions to ask regarding how the back end will operate are how elements in the virtual environment are geometrically represented, how the data describing the state of the environment is structured, and how the code that manipulates this data is given to the system.

2 Geometric Representation of Environment Elements

2.1 Introduction

Prior to rendering, every computer-rendered image is a collection of shapes. The nature of these shapes is arbitrary. An elements may be used in rendering so long as it can be positioned and configured in a space as well as have its surface mapped onto a grid for eventual write to an image. Nonetheless, different shapes have different strengths and weaknesses when it comes to the usual tasks involved with rendering an image.

2.2 Criteria

Since the system created through this project is supposed to be an environment which emulates a three-dimensional analog to painting, it is expected that users would be able to manipulate elements in the scene by sweeping some three-dimensional tool tip over a swath of space. This means that whatever elements are used should be able to represent alterations to specific volumes in the virtual environment. Furthermore, in order to match minimum expectations regarding system performance, these element alteration operations must be quick to perform.

2.3 Non-uniform Rational B-splines (NURBS)

NURBS, in essence, are patches of surfaces defined by a two-dimensional grid of splines [1][2]. For those that do not know, splines are curved line segments which have end positions and contours defined through an equation which relies upon a set of control points as parameters [1][2]. By manipulating the position of these control points, the contours of a NURBS-based surface may be warped into arbitrary shapes [1][2]. Hence, by connecting enough patches of NURBS together and properly setting their control points, any arbitrary surface may be represented [1][2].

NURBS come with a number of advantages. First, because NURBS are based on continuous linear equations, their surfaces are differentiable and hence the normal of a section of a NURBS surface is inherently encoded into the element

[2]. Furthermore, this means that NURBS do not possess a maximum resolution, and so the appearance of any NURBS patch can be accurate, regardless of scale [2]. Additionally, the functional nature of NURBS makes operations involving warping the contours of objects a quick task [2].

Of course, there are drawbacks. For instance, the more complex a NURBS surface is, the more control points it must have and the harder it is to render [2]. Plus, surfaces with non-continuous contours must use multiple NURBS surfaces to create an accurate representation [1]. Furthermore, volumetric operations with NURBS are non-trivial, as NURBS only represent the surface of an object. This means that, in order to perform such operations, the bounds of the input volumes must be calculated from the surface, the bounds of the resulting volumes must be calculated through the volumetric operation, then these new volumes must be translated back into NURBS [1][2]. Moreover, NURBS are not used as much for near real-time rendering applications compared to triangles or voxels, and so resources pertaining to their use will likely be more scarce [1].

2.4 Triangles

The only number of points guaranteed to lie upon exactly one plane in three-dimensional space is three [1]. It is for this reason that most mainstream rendering applications rely upon triangles as a means of representing environments that are to be rendered [1]. Unlike NURBS, triangles do not require processing parametric equations to establish the contours of their surface because they simply represent a segment of a plane [1]. Furthermore, given that triangles need only be represented by three points, as opposed to the many points used for NURBS, they stand as a more spartan representation of surfaces [1]. Of course, surfaces made of triangles cannot approximate smooth surfaces as easily as NURBS and also have a maximum resolution [1]. As one looks more closely at a triangle-based surface, the individual facets composing the whole become recognizable [1]. Additionally, executing volumetric operations with triangles is non-trivial. Just as with NURBS, triangles simply represent the surfaces of objects rather than their volumes [1]. Nonetheless, triangles are arguably the most widespread geometric primitive used in computer graphics, so one could expect to have the most resources to work with (such as GPU capabilities) when working with triangles [1].

2.5 Voxels

Unlike NURBS or triangles, voxels represent a piece of volume in a space rather than a piece of surface [1]. Voxels are axis-aligned cubes in a three-dimensional, regular cartesian grid [1]. The main implications of this are that locality-based operations such as volumetric addition and subtraction are easier to perform whereas operations pertaining to transformation (translation, rotation, scaling) are much more difficult to perform [1]. This is because all modifications to a voxel grid must be done by changing the values local to the affected voxels [1]. So, removing a section of an object is simply a matter of overwriting the associated voxels with a value indicating transparency. Meanwhile, moving an object requires erasing its at its current position and writing them at its new transformation. Furthermore, voxels also have a maximum resolution and do not have the benefit of triangles, which may be oriented in any direction, leaving voxel-based surfaces that are not axis aligned to look jagged, unless the voxel grid is subdivided into a small enough resolution [1].

2.6 Decision

NURBS and triangles represent only the surfaces of rendered objects, and hence inherently add complexity to performing operations which manipulate the volumes of an object [1]. With voxels, performing volume-based operations requires

only the assignment of data to the voxels which occupy the volume in question [1]. Given that the central criteria of this selection is based upon performance in volume-based operations, the clear choice in this matter is to use voxels.

3 Organization of Environment Data

3.1 Introduction

Data can be stored in a variety of ways, each of which is favors particular use cases. By selecting an organization scheme for environment geometry which favors the use cases expected of the system, one can ensure better performance.

3.2 Criteria

Given that the system to be built is meant to act as a creative environment, there are few maximum bounds one can assume about the resources required by the user. The geometries created by the user could include large and complex scenes, and so it is necessary to ensure that the way geometric elements in the environment are stored is efficient. At the same time, speed of access and manipulation is important to matching expected performance. Therefore, a data organization scheme which is efficient both in memory footprint and speed is necessary.

3.3 Three-Dimensional Array

A three-dimensional array is arguably the simplest spatially-organized 3D data structure. Accessing a specific voxel in such an array is a simple matter of referencing the element in the array with subscripts corresponding to the location of the voxel. This means that retrieval of the data in any given voxel should happen in constant time. The trade off to this is that every section of the grid in question must be represented at the same resolution regardless of actual content. This can significantly impede processes which do not simply require one sample into the space but samples from swaths of contiguous voxels. For instance, a grid may contain only one ”filled” voxel, with the rest representing only empty space, but a ray trace through the grid would nonetheless have to sample many empty voxels prior to determining whether it has hit anything or traced out of the grid. Furthermore, the manipulation of large sections of a scene would also require modifications on a voxel-by-voxel basis, regardless of the actual level of detail required by the task. Additionally, should the system be configured to remove or add elements to the data structure in response to how close the user is to them (which may be needed, should the user wish to paint in the volume that they just traveled into), every single point of culled data would have to be read out, and all remaining data would have to be erased and rewritten to their newly shifted position in the grid.

3.4 Binary Space Partition (BSP) Tree

A BSP tree recursively segments the elements in an environment using planes [3]. First, the average location of every element in the environment is found, at which point a plane intersecting this point is used to divide the environment in half [3]. Any orientation of this plane meets the requirements of a BSP, so long as it goes through this center point, although certain orientations are considered more optimal [3]. For instance, planes which do not cut through any objects make matters generally easier to manage than planes that do [3]. Once this initial cut is made, the process of finding a center point and segmenting along a plane is applied recursively on each side of this original cut [3]. Once these sections have been subdivided enough (enough meaning different criteria to different people, though this is generally some threshold involving the volume or number of objects in a section), a binary tree is derived from the segments [3]. The

organization of this tree corresponds to how the scene was subdivided, with the content of the two branches extending from the root each containing the elements from one side of the initial cut and the other respectively [3]. Likewise, branches extending from those branches have elements distributed between them corresponding to the segmentation performed on the second level of recursion [3].

The advantage of segmenting an environment like this is that it is much easier to determine which elements are contained within a swath of space by traversing a BSP tree to the sections corresponding to that swath and checking the list of elements attached at the associated leaf nodes [3]. This means sections of space not occupied by geometric elements can largely be ignored, as only non-empty content is represented in such a tree [3]. Unfortunately, BSP trees are not optimal for operating with voxels, as voxels represent both empty and non-empty spaces, unlike triangles or NURBS [1]. This means that the very space which a BSP tree is meant to remove from the representation of a scene will be represented anyway by the voxels contained within its leaf nodes. Even if each leaf node in a BSP tree stored only the minimum bounding box of non-empty voxels in its contents as a three-dimensional array, the BSP tree would not have much performance benefit over standard three-dimensional arrays except for environments with sparsely distributed clumps of geometry. Furthermore, should each leaf node store the non-empty voxels of its contents as a list, it would operate well with scenes that have very few voxels, regardless of distribution. Nonetheless, it would have increasingly poor performance for environments dense with geometry. Thus, a BSP tree offers little advantage in storing voxels, unless one assumes unreasonable constraints regarding what geometries a user is creating through the system.

3.5 Sparse Voxel Octree

An sparse voxel octree uses many of the concepts behind a BSP tree, including recursive segmentation and location-based storage in a tree [4]. It is also optimized for voxels [4].

The concept is this: The space representing an environment is an axis-aligned cube. In turn, this cube represents the root of the octree [4]. Furthermore, this cube is subdivided evenly into eight octants along each coordinate plane [4]. Each of these octants are, themselves, cubes which are represented as the child nodes of the root of the octree [4]. Should some voxel at some power-of-two resolution be written to this octree, the octant of the root cube containing that voxel will be recursively subdivided, where only octants containing the written voxel are expanded in the octree [4]. This recursion is performed until a node is added to the octree with the size and location of the written voxel, wherein the data associated with that voxel is stored [4].

Should a voxel be written to the octant of a cube which renders the content of the cube uniform (for instance all empty, or all blue), that octant and its siblings are pruned from the octree and their data is represented in the parent cube [4]. In this manner, only the resolution that is required to encode the content of a given branch is kept, ensuring that the memory footprint and retrieval time of the octree is kept low. Furthermore, the scattered nature of sparse voxel octrees makes the removal and shifting of data through the tree quicker, as shifting voxels is more a matter of shifting the pointers in the tree rather than the data the pointers link to [4].

3.6 Decision

While three-dimensional arrays likely achieve best random access speed for any given point in space, the access patterns expected of the system being built by this project do not include random read and write operations throughout a space. Instead, access and manipulation is expected to be performed in contiguous swaths of space, be they the paths of ray traces or the user's tool tip. BSP trees, while favoring access to contiguous sections of a space, generally favor access to

geometric elements which do not explicitly represent empty space, such as triangles or NURBS [3]. Of the above options, only sparse voxel octrees offer quick access to contiguous sections of voxels as well as efficiency in storing such voxels. Therefore, sparse voxel octrees will be used to store the data encoding the environment represented by the system.

4 Integration of Environment Manipulation Tools

4.1 Introduction

Given the basic definition of the project, the user must be able to manipulate the content of the virtual environment in the system. In order to do this, the system needs to give the user an intuitive means of performing this manipulation as well as provide the necessary parameters and code to the GPU to execute this manipulation upon the data representing the scene. The code that describes what a GPU does is called a shader. Shaders can be compiled dynamically and sent to the GPU for execution, however it is important to consider the model that is used for providing this code to the GPU and who would be able to add code, should the demand for more tools arise [5].

4.2 Criteria

There are a number of factors to consider when deciding how tools are integrated into the system. The first of these factors is performance, which is integral to meeting project requirements. While secondary to meeting project requirements, the extensibility of the toolset the system uses would effect how useful the system is to the public at large and hence how many people would be willing to adopt it for their work. Additionally, the time and effort required by the PolyVox team to maintain this toolset is useful to consider because, the less time and effort is required, the more the team may use for other components of the project.

4.3 Statically Defined Tools

The simplest method of providing tools to the system is to hard code them and load them at the start-up of the system. This offers a number of advantages. First, since the team working on this project would be the only ones writing the code for tools, one can make guarantees about the performance of theses tools. Furthermore, this option means that users do not have to worry about the inner workings of the system in order to use it. This being said, two obvious downsides to this course of action is that the system itself is left a lot less flexible than it could be and it forces more responsibility upon the team to ensure all necessary tools are provided.

4.4 Dynamically Defined Tools

As stated above, the shader code that informs how a GPU could manipulate the data representing an environment may be compiled while the system is running [5]. This means that users could provide their own shader code files to load into the system, enabling them to create new tools as needed. This stands as the most flexible solution, however the increased capabilities of the user means that the team can provide little guarantees about how the system can operate when using custom tools.

Those who write tool code would need to know how the system works at an in-depth level in order to be able to write effective code. Furthermore, they would be responsible for directly manipulating the data structure representing the scene, which opens up several possibilities for creating large problems for the system. Additionally, this forces the team to supply in-depth documentation of the system as well as technical support for whatever problems that may arise

from using custom code. Plus, changes in the inner-workings of the system could render some custom tools unusable in newer versions of the system.

4.5 Shader Code Injection

A middle ground between the two options is to allow users to write a limited set of functions which the system itself incorporates into template tool code that is then compiled and added to the list of tools available. While not as flexible as fully customizable tools, this allows the team to handle more of the difficult boilerplate procedures necessary to keeping the system running smoothly and does not force the user to learn about the in-depth operation of the system. Additionally, version compatibility would be easier to manage, as the team could ensure that the functions that users code could always map to whatever new functionalities are incorporated into the system.

4.6 Decision

Given that, on a broad level, most tools in this system operate the same and the operation of tools will largely be determined by code executed at small, key points of operation, it would make sense to use shader code injection. The team could benefit from operating via this method when developing tools for the system, as it would allow members to rely upon the pre-made template to handle system management. Furthermore, the option for user-defined tools offers a great deal of extensibility to the system and affords less responsibility to the team when it comes to keeping up with demand for new types of tools.

REFERENCES

[1] M. Botsch, M. Pauly, C. Rossl, S. Bischoff, and L. Kobbelt, "Geometric modeling based on triangle meshes," in *ACM SIGGRAPH 2006 Courses*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1185657.1185839

[2] D. Santa-Cruz and T. Ebrahimi, "Coding of 3d virtual objects with nurbs," *Signal Processing*, vol. 82, no. 11, pp. 1581 – 1593, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165168402003031

[3] B. Naylor, "Binary space partitioning trees as an alternative representation of polytopes," *Computer-Aided Design*, vol. 22, no. 4, pp. 250–252, 1990.

[4] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[5] T. K. G. Inc. (2017) Shader compilation. [Online]. Available: https://www.khronos.org/opengl/wiki/Shader_Compilation