Oregon State University
College of Engineering

CS Capstone Technology Review

November 14, 2017

3D Virtual Reality Painting

Prepared for

EECS

Dr. Kirsten Winters _____ _____
Signature *Date*

Dr. Mike Bailey _____ _____
Signature *Date*

Prepared by

Group 66
PolyVox

Braxton Cuneo _____ _____
Signature *Date*

Abstract

As the person 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 octree and the data will be manipulated by dynamically loaded shader code which is injected into a standard tool template which is then run on the GPU managing the system data.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 perform 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

3 Introduction

Prior to rendering, every computer-rendered image is a collection of shapes. The nature of these shapes is arbitrary. So long as they can be positioned and configured in a space and 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.

3.1 Non-uniform Rational B-splines (NURBS)

NURBS, in essence, are patches of surfaces defined by a two-dimensional grid of splines. For those that dont know, splines, themselves, are curved line segments which have end positions and contours defined through an equation which relies upon a set of control points as parameters. By the manipulating the position of these control points the contours of a NURBS-based surface may be warped into arbitrary shapes. Hence, by connecting enough patches of NURBS together and properly setting their control points, any arbitrary surface may be represented.

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 coded into the element. 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. Additionally, the functional nature of NURBS makes operations involving warping the contours of objects a quick task.

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. Furthermore, performing volumetric operations are non-trivial and require the decomposition of each NURBS surface involved into several smaller elements. Moreover, NURBS are generally not used very much for real-time rendering applications, and so resources pertaining to their use will be more scarce in comparison to those of triangles or voxels.

3.2 Triangles

The only number of points guaranteed to lie upon exactly one plane in three-dimensional space is three. It is for this reason that most mainstream rendering applications rely upon triangles as a means of representing environments that are to be rendered. Unlike NURBS, triangles do not rely upon equations to define their surface, as it is already implied by the position of its three points. Furthermore, the fact that triangles need only be represented by three points, as opposed to the many points used for NURBS, they stand as a much more spartan representation of surfaces. Of course, surfaces made of triangles cannot approximate smooth surfaces as easily as NURBS and also have a maximum resolution. As one looks more closely at a triangle-based surface, the individual facets composing the whole become recognizable. Additionally, executing volumetric operations with triangles is also non-trivial, as, just like NURBS, triangles simply represent the surfaces of objects rather than their volumes. 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.

3.3 Voxels

Unlike NURBS or triangles, voxels represent a piece of volume in a space rather than a piece of surface. Voxels are axis-aligned cubes in a three-dimensional, regular cartesian grid. 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. This is because all modifications to a voxel grid must be done by changing the values local to the affected voxels. 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.

3.4 Decision

Given that a central feature of this project is that a user working with the system should be able to do location-based manipulation such as adding, erasing, and modifying material in a limited area around their virtual tool, it would be optimal to use a geometric representation that offered the best performance for location-based operations: voxels. Additionally, as can be read below, the grid-aligned nature of voxels lend themselves to efficient data organization.

4 Organization of Environment Data

4.1 Introduction

Data can be stored in myriad ways, each of which is optimized for particular use cases. In this particular use case, rapid retrieval and modification of contents are key, the former particular for rendering and the latter particular for user manipulation of the scene. Additionally, given that the modifications a user makes will generally be centered in some area, data structures which favor the manipulation of data bounded within a target volume would be preferred.

4.2 Three-Dimensional Array

A three-dimensional array is arguably the simplest spatially-organized 3D data structure. Access into a specific voxel in such an array is a simple matter of referencing the element in the array with subscripts corresponding to the voxels location. 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 scene 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 have to perform 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 al remaining data would have to be erased and rewritten to their newly shifted position in the grid.

4.3 Binary Space Partition (BSP) Tree

A BSP tree recursively segments the elements in an environment using planes. 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. 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. For instance, planes which do not cut through any objects make matters generally easier to manage than planes that do. 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. 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. 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. Likewise, branches extending from those branches have elements distributed between them corresponding to the segmentation performed on the second level of recursion.

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. This means sections of empty space can largely be ignored, as only non-empty content is represented in such a tree. Unfortunately, BSP trees may not be considered optimal for operating with voxels, as generally more voxels are required to represent an object in comparison to triangles or NURBS. What makes voxels worthwhile to use is that, when organized along a grid, location-based access is relatively simple. Unless the voxels in each segment of a given BSP tree were sparse, it would make more sense to store them in a grid local to the segment rather than as a list. At the same time, storing voxels in a local grid per-segment, even assuming that such a grid encompassed the minimal bounding box of the voxels in that segment, would have many of the problems of a three-dimensional array (processing vast volumes of empty space, higher memory footprint, hard to shift data around) with additional processing and memory overhead.

4.4 Octree

An octree uses many of the concepts behind a BSP tree, such as recursive segmentation and location-based storage in a tree, and is also optimized for voxels. The concept is this: the space representing an environment is an axis-aligned cube. This cube represents the root of the octree. Furthermore, this cube is subdivided evenly into eight octants along each coordinate axis. Each of these octants are, themselves, cubes which are represented as the child nodes of the root of the octree. 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 tree. This recursion is performed until a node is added to the tree with the size and location of the written voxel, wherein the data associated with that voxel is stored.

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), it and its siblings are pruned from the tree and their data is represented in the parent cube. 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 tree is kept low. Furthermore, the scattered nature of 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.

4.5 Decisison

Given the advantages which octrees represent in comparison to three-dimensional arrays and BSP trees when it comes to the storage and manipulation of voxels, particularly with only using the minimally required resolution for included elements, octrees shall be used for representing environments in this projects system.

5 Integration of Environment Manipulation Tools

5.1 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. Statically defined tools

5.2 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.

5.3 Dynamially 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. This means that users could provide their own shader 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.

5.4 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 custom 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.

5.5 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 go with the third option. The team could benefit from operating via a template structure when developing tools for they system, as it would decrease workload when developing new tools. 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. [1] [2] [3] [4]

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] 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.

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