Next Generation Content Creation: An Investigative Approach

by

Nicholas Vining
B.Sc., University of Victoria, 2009

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Nicholas Vining, 2011
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part,
by photocopying or other means, without the permission of the author.

Next Generation Content Creation: An Investigative Approach

by

Nicholas Vining
B.Sc., University of Victoria, 2009

Supervisory Committee

Dr. Bruce Gooch, Co-Supervisor
(Department of Computer Science, University of Victoria)

Dr. Bruce Kapron, Co-Supervisor
(Department of Computer Science, University of Victoria)

Supervisory Committee

Dr. Bruce Gooch, Co-Supervisor
(Department of Computer Science, University of Victoria)

Dr. Bruce Kapron, Co-Supervisor
(Department of Computer Science, University of Victoria)

ABSTRACT

The rising cost in video game content creation, both in terms of man hours and in terms of monetary dollars, restricts the ability of video game developers to create unique, entertaining content. Motivated by how this cost is a direct result of "next-generation graphics", I am motivated to ask: what would a next-generation content creation tool look like? I investigate the problem by constructing several such tools. In particular, I construct a mesh quilting algorithm for random level generation, a rapid level construction toolkit based on the concept of an architectural blueprint but supporting features such as complex silhouette geometry and roof geometry, and a tool for rapidly painting world textures. I also introduce a new system for accessing barycentric coordinate data from within the fragment shader, which can be used in support of real-time 3D image quilting, more accurate normal interpolation, and texture rendering from within the world painting tool. Some history of video game content creation is discussed, and a roadmap is charted for future development.

Contents

Supervisory Committee ii

Abstract iii

Contents iv

List of Algorithms vi

List of Figures vii

Acknowledgements ix

Dedication x

1 Introduction 1
- 1.1 Contributions 4

2 Background 5
- 2.1 The Game Development Industry 5
- 2.2 Level Construction 5
- 2.3 Level Texturing 10

3 Procedural Content Generation 13
- 3.1 Model Synthesis from Exemplars 13
- 3.2 Model Quilting 16
- 3.3 Comparison of Model Quilting and Model Synthesis 18
- 3.4 Rule-based Content Generation 18

4 Level Modelling **22**
4.1 LevelShop 22
4.2 Planar Sweep Algorithm 26
4.3 Weighted Skeleton Systems 28
4.3.1 Weighted Skeleton Systems and the Motorcycle Graph 36
4.3.2 Efficient Parallelization of Level Creation 41
4.4 Procedural Extrusion and Model Synthesis 42

5 Content Creation for Sparse Virtual Texturing **44**
5.1 Background and Related Work 46
5.2 Mesh Colors 47
5.2.1 Review of Mesh Colors 47
5.2.2 Practical Real-Time Implementation 48
5.2.3 Virtual Paging for Mesh Colors 50
5.2.4 Mesh Preprocessing 52
5.3 Results 53
5.4 Baking Parametrization-Free Virtual Textures 54

6 Applications of Barycentric Coordinates **57**
6.1 General Per-Face Data 57
6.2 Barycentric Coordinates in the Fragment Shader 59
6.3 General, Per-Face Data, Second Approach 63
6.4 Applications 64

7 Conclusion **66**

Bibliography **68**

A Triangulations of the Disc with all Interior Vertices of Even Degree are 3-Colorable **73**

B Shader Code **76**
B.1 Parametrization-Free Virtual Texturing Shader 76

List of Algorithms

3.1 Merrell's Discrete Algorithm for Model Synthesis. 15
4.1 Planar Sweep implementation of LevelShop. 29
4.2 Kelly and Wonka's procedural extrusion algorithm. 33
4.3 Continuous Model Synthesis algorithm. 43
6.1 3-coloring a triangulation of the unit disc. 62

List of Figures

2.2.1 *Hammer, the editor for the Source Engine, in action. The upper left screen displays a preview perspective view of the map; the remainder of the user interface is devoted to brush creation and editing in three orthogonal perspectives. Image courtesy Valve Software.* 7
2.2.2 LittleBigPlanet's Create Mode lets users create their own levels. Sophisticated constructions such as this scene are possible with nothing more than a set of 2D extrusions and a "paint mode" based on constructive solid geometry. Image (C) Media Molecule. 8
2.3.1 Megatexturing in the video game RAGE allows for vivid, detailed external views with unique surface detail. Image (C) id Software. 11
2.3.2 An area of poor texture quality in RAGE. Also note the artifact on the floor caused by incorrect bilinear filtering across a page boundary in the virtual texture. Image (C) id Software. . 12

3.1.1 Left: Input Set: Right: Output from our implementation of Merrell's Algorithm for Discrete Model Synthesis. Note that the algorithm preserves locality of structure on the tower piece, but fails to correctly capture our intuitively specified requirements about the walls. 15
3.2.1 Mesh quilting produces recognizeable local patches of content, but no connected structure. 18
3.4.1 Dungeons of Dredmor. Image courtesy of Gaslamp Games. . . 19

4.1.1 LevelShop in Action. Beige areas are floors. Red areas are portal connectors; blue areas are floor connectors, and green areas are ramp connectors. Courtesy of Harvey Fong. 23
4.1.2 Output from LevelShop. Note that the geometry is rough and unfinished. 24
4.3.1 The straight skeleton of a polygon. 30
4.3.2 The straight skeleton of a polygon can be computed via edge collapse and edge split events. 31
4.3.3 A small collection of houses. 37
4.3.4 The Motorcycle Graph subdivides a region into convex subregions. Image after Stefan Huber. 39

5.2.1 Mesh colors are distributed evenly across the surface of a triangle. Left to right: a triangle at $R = 1$ consists only of vertex colors; a triangle at $R = 2$ contains color data at both vertices and edge midpoints; a triangle at resolution $R = 4$ contains vertex, edge and face colors. 48
5.3.1 Parametrization-free virtual texturing in action. 54

6.1.1 The triangular graph T_4 does not have a bipartite matching between its vertices and faces. 58
6.2.1 Left to right: a) A disc to be colored, containing an interior node with odd degree. b) I apply our cutting algorithm in order to make this node even degree. c) The resulting disc can then be 3-colored using a simple, greedy algorithm. 62

Acknowledgements

With thanks to the usual suspects:

- Dr. Bruce Kapron, who went to an auction and came home with a graduate student. (Auctions are dangerous like this; you see things, you think it's a good idea at the time...)
- Dr. Bruce Gooch, for promptly stepping in when the money ran out.
- Dr. Sasha Fedorova and Dr. Peter F. Driessen, for stepping in to be an outside member and a defense chair at the last possible moment.
- Dr. Alla Sheffa, for actually making me shut up and finish this.
- Wendy Beggs, who does not get nearly enough credit for her role in getting grad students out the door, and who has put up with more of my nonsense than any other human on the planet.
- Thomas Kelly, Harvey Fong, Cem Yuksel, and Alex Evans for answering questions about their work, and for giving so generously. Like Newton, I stand on the shoulders of giants. Unfortunately, the resemblance ends there.
- Dr. Jing Huang, Dr. Gary McGillivray, and Dr. Peter Dukes for attempting to teach me combinatorics back in the heady days of my undergraduate degree. Who knew that any of it would sink in?
- The usual suspects: Matthew Skala, Nick Alexander, Daniel Jacobsen, David Baumgart, Ben Kyles, Ashlin Richardson, Kevin Stuart, Sean Barrett, and Micah J. Best.
- and Esme Cunningham, for, amongst other things, baking cookies and listening to me complain.

Dedication

In memory of John R. Hall; a brilliant mind lost too early.

Chapter 1

Introduction

Academic papers and video games both use the term "next-generation graphics" to discuss their contributions to the state-of-the-art. Hardware manufacturers advertise themselves as selling "next-generation graphics cards" or a "next-generation console"; academic researchers discuss "next-generation lighting algorithms" or "next-generation texturing algorithms." This term conveys meaning to the reader: the author, or manufacturer, wishes to convince us that his work is a leap above the current *de facto* standards for rendering technology in terms of visual fidelity, realism, or efficiency. This term, "the next generation", is then leveraged by the computer game development industry in order to promote their products. Advertisers talk about the "current generation" of video game consoles, and compare and contrast them with the "next generation" of video game consoles as soon as they are announced. The implication, at each state, is that the next generation promises a better return on the customer's investment than the first.

Every advance in visual fidelity in real-time computer graphics is coupled with an associated increase in cost, in terms of money and in terms of artist time. In particular, as the complexity of a graphics engine increases, the amount of time it takes to produce content for the game increases. Major shifts in technology over the past decade have included:

- the transition from two-dimensional to three-dimensional graphics,
- the transition from lightmaps to real-time lighting,

- the transition to higher-resolution, photo-sourced textures,
- the transition to normal mapping, including the compilation of normal maps from higher-resolution versions of low-resolution maps,
- the use of displacement maps,
- high dynamic range rendering

to name but a few.

With each of these advances, a game development studio faces a very real escalation in terms of cost and time. For instance, the amount of time that it takes to produce a level has increased dramatically. For instance, consider the video game *Wolfenstein 3D,* the original first-person shooter. In the book Masters of Doom[28], the developers at ID Software noted that a Wolfenstein 3D level took a day, on average, to create. By the time ID Software's next project, *DOOM*, was under construction, the average time to build a level had jumped from a day to a week. These figures are for a single developer. Modern levels require teams of multiple developers with different skill sets, working in tandem, to build; the average level takes between three and six months to be completed.

This situation is widely considered to be unsustainable. At the start of the last decade, a AAA title would cost on the order of 1.5 to 2 million dollars.[20] In 2011, it is harder to get correct statistics as most developers no longer self-report in industry publications, but the game publisher Ubisoft has reported as recently as 2009 that a AAA title costs approximately 30 million dollars to make, and that they expect the next generation of titles on future platforms to cost upwards of 60 million dollars.[41] This trend suggests that costs of game development approximately double every two to three years, in a parallel with Moore's Law.

In fact, the consequences associated with AAA development are one of the factors driving the game industry towards a new model of independent game development, and are causing major studios to explore the realm of social media platforms. Games for platforms such as Facebook or the iPhone can be built at a fraction of the cost of a AAA title for the Playstation 3 or XBOX 360, and the reward-to-investment ratio is significantly less volatile.

Additionally, any given project involves less risk: publishers are less prepared to sink the millions of dollars needed to build a AAA title into a property that they do not know will be a hit. This leads to a plethora of "safe bets" and sequels based on existing properties from decades ago. Developing a new AAA property is widely considered to be incredibly difficult.

Increased development times beg the question: if we have "next-generation graphics", what does "next-generation content creation" look like? The academic community is in a good position to address this question, in a manner that the commercial game development community cannot. Tools, to the game developer, are an afterthought; something to be lashed together hastily and without an attempt to improve on the efficiency of the process. The basic formula for level creation for a video game has not changed from the early editors for *Wolfenstein 3D* and *DOOM,* pioneered during the early nineties.

Content creation tools have not scaled, in terms of their efficiency, at a rate that is in line with next-generation graphics. If costs continue to increase quadratically, without a way of keeping things in check, game developers may face a situation where they can no longer build video games that take full advantage of the hardware and software opportunities presented to us by the next generation of video cards and gaming consoles.

The purpose of this thesis is to attempt to build next-generation tools for content creation. It is unclear what these tools look like, or what they should do. I will focus our efforts on level creation: the process by which a designer produces a level, or a map, for a video game. I wish to make the construction of this map as efficient as possible, and as flexible as possible: artists should be able to build levels quickly, and in a way that is adaptable to changing demands and designs. If game development is an art, artists have been fighting for years with oil paint, and our goal is to give them modern acrylics.

To illustrate our concept for a next-generation level editor, I have produced a sequence of tools for designing a level, modeling a level's geometry, and then painting the level. These tools have been built based on our experience over the past decade of work in the video game industry, and as the owner of a small game development studio.

1.1 Contributions

This thesis contributes the following:

- A new system for the creation of video game levels from a 2D map and a set of guidelines. Our system allows for the construction of rich, detailed geometry, including doors and windows, and is robust and numerically stable.

- A new system for the creation of video game levels from an example geometry by "mesh quilting."

- A new system for painting textures on level geometry. Our system does not require artists to assign texture coordinates to level geometry, and creates seamless color data (and, if desired, normal data.) Additionally, our system can bake out to one or more texture maps, and is adaptive - texture memory is allocated and prioritized using an approach based on signal-specialized parametrization techniques.[44, 47] Since this analysis is performed at bake time, artists can simply paint without having to prioritize texture information.

- A new algorithm for sparse virtual texturing ("Megatexturing")[4] that eliminates the seams found in conventional implementations,

- A new system for accessing barycentric coordinates and per-face information in geometry on DirectX 9 level hardware, with applications towards surface normalization and texture splatting.

Chapter 2

Background

It is informative to consider how other games and game engines have tackled the problems of level development, as well as how level construction and painting has been represented in academic works.

2.1 The Game Development Industry

2.2 Level Construction

The standard reference on constructive solid geometry algorithms is the sequence of works by Thibault and Naylor.[48, 42] The basic approach that they outline in their seminal paper converts polygonal meshes to a BSP-tree based representation. Performing a CSG algorithm - union, intersection, or difference - on the two meshes is then equivalent to peforming an operation on the two BSP trees and converting back to a polygonal format.

The actual use of BSP trees for level geometry in video games is well known. The practice originated with DOOM by id Software, where John Carmack used a two-dimensional BSP-tree based algorithm and a polar coordinate based raycaster to rapidly determine visibility in the 3D world. The followup game, Quake, extended this to three dimensions[28], and this would be the dominant paradigm for video game engines for the next ten to fifteen years.

Owing to the influence of Quake, many video game engines - for instance, Valve Software's Source engine - still use BSP-tree based systems for level editing. Levels are constructed from convex primitives known as *brushes.*

The BSP-based representation of level geometry is also convenient for CSG operations. BSP-based level geometry can be intersected, unioned, and differenced with itself without a conversion from a mesh based to BSP based format.

Correctly compiling a level requires a developer to ensure that this collection of brushes forms a water-tight mesh. Failure to provide watertightness - a condition that must be specified by hand! - breaks the level. Because modern video game geometry is so finely detailed, most game engines eschew the use of BSP trees for visibility, preferring instead to use portals and occluders, hybrid systems such as CHC++[36], or various quadtree and octree-based schemes which are more appropriate for outdoor terrain. This eliminates certain restrictions on level creators, most notably the onus that a level must be "watertight" - a closed mesh with no holes. Modern level creation now consists of a two-stage process:

1. A "blocking in" stage, where crude level geometry is built to prototype gameplay,
2. A "detailing" phase in which the crude level geometry serves as a skeleton and framework for more refined, artist-generated content.

The best example of an engine that is built around this development paradigm is the Unreal Engine's UnrealEd editor. An artist lays down BSP brushes, then places high-resolution meshes authored in an external program such as 3D Studio Max or Maya, overtop of the BSP brushes.

The disadvantage to this approach is obvious: if a level creator wishes to change direction after the "blocking in" stage, he must rebuild all of his detailed, high-polygon geometry. This often means throwing out months of work. One of our goals is to address this problem, and I will propose several solutions in Section 4.

CSG-based editors also show up in other video games. The video game LittleBigPlanet, by UK based developer Media Molecule, integrates level con-

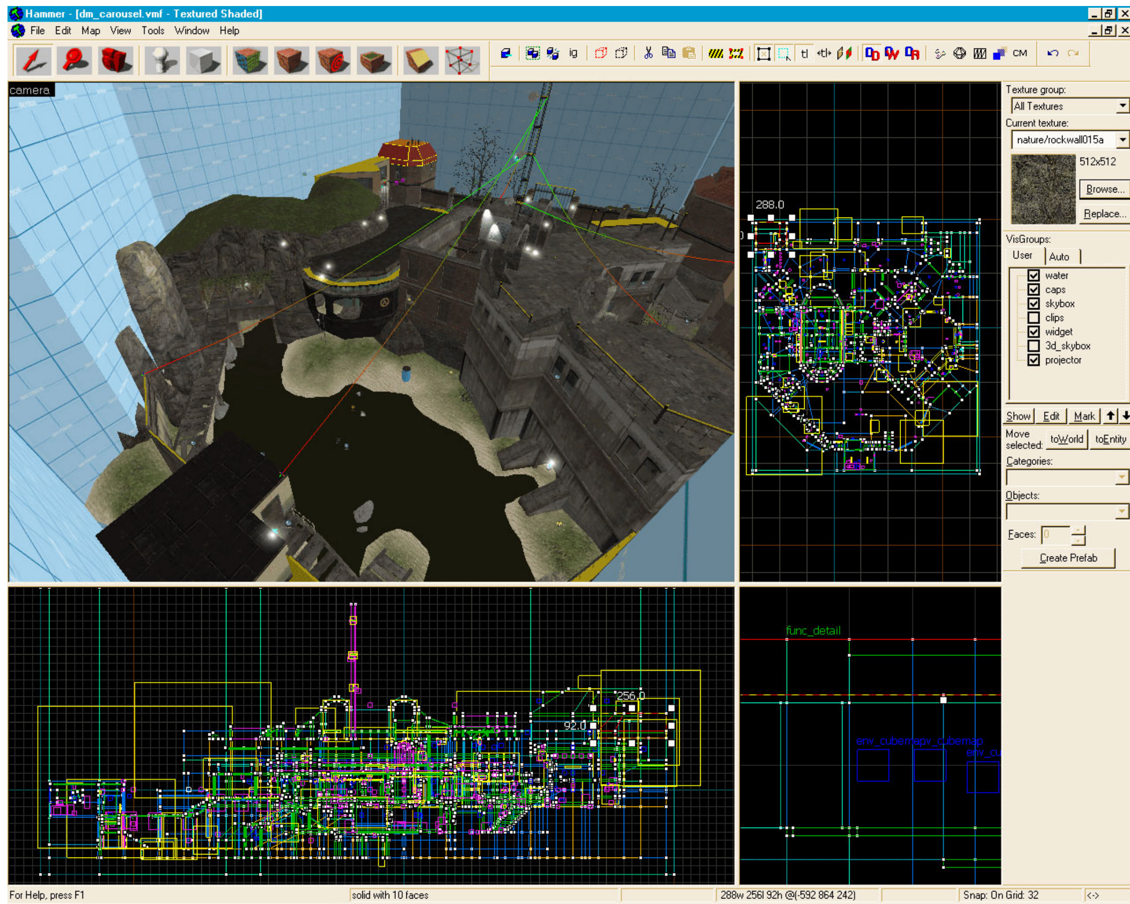


Figure 2.2.1: Hammer, the editor for the Source Engine, in action. The upper left screen displays a preview perspective view of the map; the remainder of the user interface is devoted to brush creation and editing in three orthogonal perspectives. Image courtesy Valve Software.

Figure 2.2.2: LittleBigPlanet's Create Mode lets users create their own levels. Sophisticated constructions such as this scene are possible with nothing more than a set of 2D extrusions and a "paint mode" based on constructive solid geometry. Image (C) Media Molecule.

struction into the game itself. In fact, the "Create" mode of LittleBigPlanet offers users the same tools that were used by the developers themselves. Using a series of shaped brushes, users can paint material of various thicknesses in a three-dimensional world along the XY plane. This is also accomplished by constructive solid geometry, although the exact mechanisms involved are not disclosed to the public. What is notable about the implementation of CSG in LittleBigPlanet is its remarkable stability. Users never have to worry about their geometry being water tight, and the game never suffers from floating point precision issues in any noticeable fashion.

The success of LittleBigPlanet in shipping a game that contains real-time constructive solid geometry is indicative of a key point that I wish to exploit, and that has not been explicitly stated, but rather implicitly understood. It is the following:

Proposition. *Constructive solid geometry systems that operate in two dimensions are inherently more stable and easier to implement than those that operate in three dimensions.*

This is a fairly obvious statement. An experienced graphics programmer will note that nearly everything is easier in two dimensions. In the case of constructive solid geometry, however, I can justify this statement by considering the linear algebra behind these computations.

Consider two volumes - one of which is defined by a collection of line segments in two dimensions, and one of which is defined by a collection of planar faces in three dimensions. If I wish to determine whether or not two line segments intersect, there are three possibilities: the line segments intersect at a point, co-linearly, or not at all. More generally, if two lines of infinite length in two dimensions intersect, they are either parallel, co-linear, or intersect at a point.

The analogous situation with three planes is more messy. Two planes in three dimensions are either parallel, the same plane, or intersect to form a line. Three planes can intersect at three lines, two lines, a point, one line, or not at all. Any programmer wishing to work with solid volumes bounded by planar half-spaces must constantly check to see whether or not he is encountering one of degenerate cases, and must handle it appropriately.[45]

There is the additional issue of numerical stability. Determining the intersection of two lines is equivalent to determining the solution of a system of linear equations in two dimensions. Determining the intersection of three planes is equivalent to determining the solution of a system of linear equations in three dimensions. Most algorithms for intersection of geometric primitives - for instance, those given in [45], rely on such methods at some point in the algorithm. Typically, such algorithms eschew Gaussian solvers for methods based on Cramer's rule and the expansion of cofactors. Cramer's rule is notoriously unstable, even for 3x3 matrices; a programmer can always reduce the numerical instability of a system by reducing the number of dimensions.

Any modelling system that I design would be well-advised to follow this principle. As I will soon demonstrate, the modelling system proposed in Section 4 does exactly that.

Most of these statements apply to interior geometry only. Exterior geom-

etry - terrains - are generally created using a heightmap and a paint-style authoring program. More complex terrain can be modelled using a voxel-based scheme such as the one outlined in work by Eric Lengyel [32]. Lengyel's basic approach, using a marching cubes based algorithm[33], is used to polygonize a voxel-based terrain, which is then textured using cube maps to resolve the problem of terrain unwrapping. I will investigate the problem of texturing terrain uniquely in section 6.

2.3 Level Texturing

Level texturing is the process by which texture information - from a series of two-dimensional images - is mapped onto created level geometry. The first presence of a texture-mapped level in a video game is widely believed to be *Ultima: Underworld.*[28]

The general workflow for texturing is simple. Level geometry - created by any of the methods in Section 2.1 - is assigned a texture map and a set of index coordinates. Surfaces can have multiple texture maps associated with them - for instance, one texture map may hold surface albedo detail, and another texture map may contain normal mapping information for use with bump mapping.

Very little work has been done in the area of automatic texture assignment. One interesting line of attack by Lefebvre and Hoppe[30] uses a texture synthesis method for arbitrary surfaces based on exemplars. This work is patented, and so I will not discuss it. Disney gave a talk on their implementation of appearance-space texture synthesis for the film *Tangled* that may circumvent the patent.[14]

Chadjas, Lefebvre and Stamminger[9] propose an intriguing algorithm for assigning textures and materials to a level of a video game. Their approach uses a surface similarity measure to determine which texture from a texture library a given surface should have. Multiple selections for materials, made by an artist, propagate throughout a level. This represents a considerable savings and is compatible with all of the work done in this thesis. The disadvantage to this approach is a lengthy pre-processing stage, although this

Figure 2.3.1: Megatexturing in the video game RAGE allows for vivid, detailed external views with unique surface detail. Image (C) id Software.

can be accelerated by using GPU-assisted raytracing.

The most recent advance in texturing has been a series of innovations known as sparse virtual texturing.[4, 23, 43] Also known by the industry buzzword "Megatexturing", sparse virtual textures allows artists to uniquely texture surfaces in a level by indexing one massive texture - of size $2^{18} \times 2^{18}$ or larger. The texture is split into pages, and a virtual cacheing mechanism is implemented using a two-stage feedback pass. In the first stage, a "visibility buffer" is rendered to determine which surfaces in the world are visible, and with what texture coordinates; the texture is then paged from disk into a page table. In the second stage, the actual correct texels are sampled from the page table using a two-stage lookup and render process. Support for bilinear filtering and mip-mapping is complicated, and requires clever math - and a certain amount of padding the page table.

The primary issue with sparse virtual texturing - other than the lack of hardware support - is that it is very difficult to author a very large texture. The texture must be assigned to the entire level in such a way that detailed areas receive more texture information than less detailed areas, and to ensure that there are no areas that the player will encounter where sufficient texture detail has not been correctly assigned.

Taking advantage of the unique detail prevalent in a single large texture

Figure 2.3.2: An area of poor texture quality in RAGE. Also note the artifact on the floor caused by incorrect bilinear filtering across a page boundary in the virtual texture. Image (C) id Software.

presents its own issues. For the video game RAGE, id Software developed a three-stage method. In the first stage, texture data is assigned normally to the level using standard assignment or brush-based techniques. In the second stage, decal-based stamps are applied. Stamp information is stored in a vector format, so that it can be baked at the last minute. Finally, the texture is baked, packed, and compressed. Some versions of RAGE suffer from severe image compression artifacts owing to the necessity of trying to fit the game content onto multiple DVDs. Additionally, it is not clear that the packing and compression can be accomplished on consumer hardware. The actual packing and compression seems to mainly involve brute-force algorithms; this is not surprising as the problem of packing multiple texture sub-elements in a texture in the most efficient texture is strongly NP-complete.

I will discuss the problem of authoring texture content for sparse virtual texturing in Chapter 5.

Chapter 3

Procedural Content Generation

Much ado has been made of the potential of procedural content generation. Touted as being able to eliminate artists entirely, the reality of procedural content generation has yet to live up to the hype. Some success has been found in the use of procedural content generation for specific, restricted content types - dating, for instance, as far back as the video game *The Elder Scrolls: Daggerfall*, where entire towns were procedurally generated from a basic algorithm. Newer examples include the unfinished videogame *Subversion* by Introversion Software, and *Dwarf Fortress* by Bay 12 Games (the latter operating using ANSI characters, and hence not particularily relevant to this thesis.)

This raises an interesting question for our next-generation content creation toolchain: is it necessary to have an artist involved in the process at all when creating a level? Can a computer simply create a level from a set of chosen example pieces? This chapter attempts to investigate this question.

3.1 Model Synthesis from Exemplars

I am motivated to consider what I will call the *synthesis by example* problem. Our ideal system takes, as input, an arbitrary artist-generated example model - without making assumptions about what it represents - and generates new content, be they models or textures, from this example model. The example model is usually referred to as an *exemplar*. In the two-dimensional

space, so-called *image synthesis* methods (many millions of citations) have produced excellent results.

In three dimensions, Paul Merrell's Ph.D thesis contained the most successful attempt at procedural model generation from an exemplar set to date. Merrell called his concept *Model Synthesis.* His first paper[37] took an exemplar that is manually split by an artist into unit regions - similar to a tile-based scheme of the sort used in two-dimensional video games, but in three dimensions - and deduced a set of rules based on an artist's example placement. His later papers expanded on this idea, allowing for low-resolution, non-tile geometry. Merrell's second paper[38] overlaid key feature planes from low-resolution geometry in an infinite grid, and polygons are allowed to "jump" from one plane to another depending on the rules defined by the exemplar. Both algorithms rely on the notion of a similarity constraint, which is used as a tie-breaker when deciding which of multiple rules to follow in order to produce results that more closely obey the exemplar.

I have focused my attention on Merrell's first algorithm, as his second algorithm is unsuitable for detailed geometry. In particular, it falls down when you have high resolution models, as each new face normal adds a new plane and a new set of constraints.

A more precise description of Merrell's algorithm is reproduced below.

I implemented Merrell's discrete algorithm and obtained some interesting results.

Merrell notes that there are some problems with his algorithm. The biggest problem with Merrell's papers - both the tile-based and plane-based approaches - is one of computational intractibility. Given a configuration of unit tiles, the problem of determining whether or not the tiles obey all the rules present in the exemplar is NP-Complete, achieved via a reduction from PLANAR 3-SAT.[1]

[1] The PLANAR 3-SAT problem asks whether or not a given a Boolean formula with three literals per clause, that can be put into a planar graph, is satisfiable. It turns out that the extra restriction on the class of formulas is useless; PLANAR 3-SAT is equivalent to 3-SAT.

Algorithm 3.1 Merrell's Discrete Algorithm for Model Synthesis.

1. Let M_0 be a simple consistent model (for instance, the empty label), and let $M = M_0$.

2. Choose a set of vertices of the model B to modify.

3. Create a new model M' where the labels for the vertices contained in B are no longer assigned.

4. While the set of new, candidate labels for M' is non-empty:

   (a) Choose a vertex and assign a label to it from the candidate labels from that vertex.

   (b) Update the set of candidate labels from M' by eliminating those labels that are no longer consistent with the past vertex assignment.

5. Set $M = M'$ and repeat if desired.

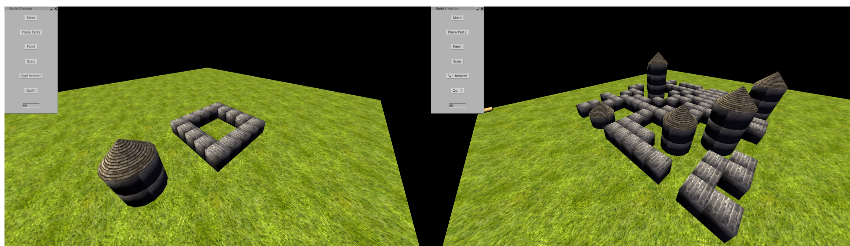


Figure 3.1.1: Left: Input Set: Right: Output from our implementation of Merrell's Algorithm for Discrete Model Synthesis. Note that the algorithm preserves locality of structure on the tower piece, but fails to correctly capture our intuitively specified requirements about the walls.

An undocumented side effect of his algorithm is that obtaining correct results - i.e. results that obey the rules that the artists want, as opposed to the rules that the exemplar set may unwillingly provide - requires correct use of tilings. For instance, consider the problem of generating castle walls. The following results are produced by our implementation of Merrell's first paper.

Castles are constructed using one, two, or four different wall types, connected to a series of rounded corners. If I use one wall brick, the resulting structure has the appearance of random noise, and bears no resemblance to a castle wall. If I use four different wall types, nothing happens - the rules are now too constricted. Only if I use two wall types do I achieve the sort of castle results that I am looking for.

There are two possible ways to address this problem. One strategy to address this problem is by making rule-creation explicit, by allowing the artists to define rules explicitly, but I have not pursued this. Another startegy is to look at how two-dimensional texture synthesis algorithms, which are more refined, have worked around these problems.

3.2 Model Quilting

Merrell's work drew inspiration from various schemes for two-dimensional texture synthesis. Most algorithms that work per-pixel have now been discarded in favour of "image quilting approaches" where random patches of texture are sampled, overlaid on each other, and run through a seam adjustment pass.[13] This approach is easily extended to model synthesis to produce "model quilting." Merrell himself noted that his approach "has more in common with the local versions" of image synthesis; it seems instructive to see what results can be obtained with global model synthesis.

Our approach is extended from the work of Efros and Freeman, who used a quilting method to synthesize two-dimensional textures. In his method, random patches of an exemplar texture are drawn on top of each other; similarity constraints are then applied to fix the borders of the quilt. The key observation of Efros and Freeman was that pixel-based methods result in correllated neighbouring pixels, but fail to provide patches of global similarity.

The same is true of Merrell's discrete algorithm; even Merrell's "similarity constraint" is not enough to truly enforce global feature resemblance. That said, it is easy to build a quilting system on top of Merrell's established framework, to produce a new form of model synthesis that uses a quilting mechanism and operates in three dimensions instead of two.[2]

Why should we believe that this new approach will produce better results than Merrell's algorithm? Consider the nature of the models that we are trying to synthesize from examples: man-made surfaces with structure! In the case of pixel-based approaches for image synthesis, the best results are always obtained with stochastically random input patches.

Let us briefly review the algorithm of Efros and Freeman.

The image quilting algorithm starts by placing an initial seed patch somewhere on the texture, chosen at random. New patches are "splatted" overtop of the existing patches, repeatedly; each patch is selected to try and be as similar to the existing patches as possible. A quilting "seam" is determined to join the two patches by computing the path from one edge of the patch to another edge of the path in such a way that texture error is minimized.

How does this algorithm generalize to three dimensions?

I implemented a naive form of mesh quilting that simply selects blocks of a suitable size - say, 5x5x5 - and draws them on top of the other. Simply splatting meshes at random gives good results, especially when coupled with a repair algorithm using the validation logic used in Merrell's code.

[2]While not used for general model or level synthesis, Zhou et al. [cit.] describe an application of "mesh quilting" for the purposes of producing a synthetic construction on a textured shell; for instance, to create a suit of chain mail armour. They do not discuss the general problem of constructing new models from old, nor do they consider applications of their work for level design. Their approach also requires a distortion-minimizing multiple-chart texture atlas.

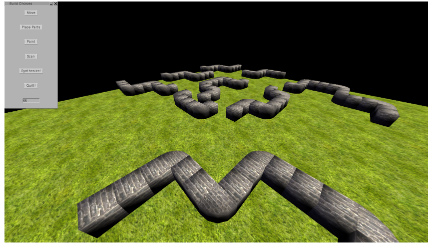


Figure 3.2.1: Mesh quilting produces recognizeable local patches of content, but no connected structure.

3.3 Comparison of Model Quilting and Model Synthesis

While Model Quilting does produce more interesting result than Model Synthesis and does not contain hidden "rules", it has some disadvantages when compared to model quilting. In particular, model quilting is truly random: there is no way to specify that you want, say, a castle with a certain configuration. Instead, the global structure of the world is dictated purely by the quilted blocks. Model synthesis, on the other hand, makes this easy: simply start with an empty, consistent model consisting of empty labels and only remove the labels from those parts of the model where you want castle geometry.

In the original paper by Efros and Freeman, the authors implement something which they call "texture transfer": mapping data from an exemplar onto a target image. It is not immediately clear how to map three-dimensional tiles on to Efros and Freeman's quality function in a way that makes sense to an artist.

3.4 Rule-based Content Generation

As a coda to this section, I give an example of a rule-based system for level generation that I have worked on. The video game *Dungeons of Dredmor* uses two forms of procedural content generation in two dimensions. These methods could extend to three dimensions, and as such the work completed

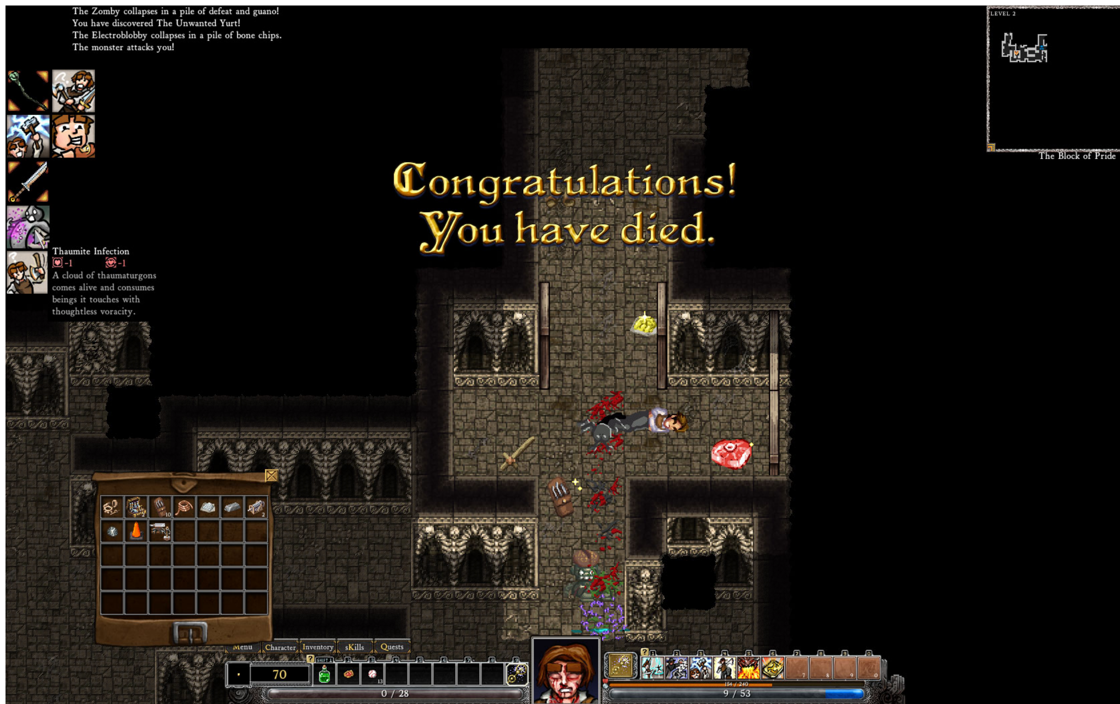


Figure 3.4.1: Dungeons of Dredmor. Image courtesy of Gaslamp Games.

for *Dredmor* warrants discussion.

Dredmor uses two rule-based systems for procedural content generation. The first system creates the dungeon layout by reading a selection of rooms from a database and deploying them around the level. Rooms are tagged with doorways, and this is the only allowed connection point between rooms.

The algorithm that *Dredmor* uses for level creation uses a basic random approach. I begin the construction of a *Dredmor* level by placing a seed room on the level. (Some Dredmor levels require the game to choose from a number of initial seed rooms; this ensures that we can successfully create large set pieces.) Untagged doorways are added to a list. Ithen grow the dungeon by attempting to randomly assign rooms from the database to un-matched doors. A room is allowed to spawn in place if it will not overwrite any existing level geometry. Once a new room is placed, I add its doorways to the list of potential doorways, and the algorithm continues. If a room cannot be

placed within a certain number of tries for a given doorway, I simply close that doorway. After all doorways are closed, I throw out the level and start again if the level does not satisfy our requirements: for instance, I require that a certain percentage of the potential tiles in the level are non-empty, and that a certain number of rooms have been placed.

This algorithm is quick and fast, and provides a modicum of artist-based control. It has two main issues. The first main issue is that it cannot generate a level whose graph is anything other than a tree. Dredmor levels inherently contain no loops. This can be monotonous from the perspective of the player. The second issue with this form of level generation is that the player will only see a set collection of rooms in his playthrough experience. We could remove this elimination by combining the room database with an approach similar to our mesh quilting algorithm; the room database would then be used as a sequence of exemplars, and rooms could overwrite themselves at locations other than doors. At the same time, this would change the way in which *Dredmor* is played: as it stands, players often use the doors as chokepoints to defeat incoming monsters. Procedural content generation must also ensure that the end level conforms to the game designer's goals.

Dredmor's second rule based system is responsible for converting the level geometry to a tile-based representation. It uses an explicit set of rules that analyzes the tiles connecting water and floor, and the tiles connecting walls and floors, to manually assign borders from a set of artist-constructed tiles. In practice, this system has been prone to breaking as the random level generator produces a wall or water configuration that was not anticipated by the artist. A given water tile can be surrounded by two hundred and fifty six different combinations of floor and water tiles. We can, of course, reduce the size of the computation: in particular, we can compute the number of actual possible configurations by the use of Polya's enumeration theorem.[22] Similar approaches are employed in three dimensions in the form of the celebrated "marching cubes" algorithm.[33] Nonetheless, it is instructive to consider just how much trouble an artist can get himself in via a naive, rule-based system.

Comparing the output of *Dredmor*'s level generator to any approach based on either model generation algorithm is instructive. While the model generation algorithms can produce interesting 3D models, none of them are capable of producing an actual level with a guarantee of structure. I am reluctantly

forced to conclude that this technology is not ready for prime time. A more promising approach is building a skeleton of a level using a rule-based approach, and then decorating it with model synthesis - either of the quilting variety, or of the model generation variety. Merrell's algorithm makes this very easy; structure can be imposed on a generated mesh by restricting the initial set of candidate labels. For a more practical approach, however, it is necessary to consider something else.

Chapter 4

Level Modelling

In the previous chapter, I considered the state-of-the-art in level synthesis via procedural methods. In this chapter, I will consider how a next-generation content creation tool that expands on the traditional modelling process may be created.

4.1 LevelShop

The traditional level editing process, as described in a leading textbook, involves the construction of a level using so-called "brushes" - a term that dates from the era when levels in video games were internally stored as BSP trees. In a brush-based editor, convex polygons are defined in three-dimensional space using a standard, three dimensional modelling interface that presents the user with the ability to draw boundaries of polygons in orthographic projected views.

The use of convex polygons is a convenient holdover from the days of BSP based work. Since most level elements are not convex, and in fact require a higher level of detail than can be conveniently assembled in a crude level editor, modern level construction consists in practice of assembling a number of pre-fabricated elements, produced by a prop artist in an art package such as *3D Studio Max* or *Maya*, and gluing the result together in the level editor. The level designer's ability to construct a level that corresponds to his vision of gameplay is limited by the collection of prefabricated widgets available to him. He is detached from the artistic creation process.

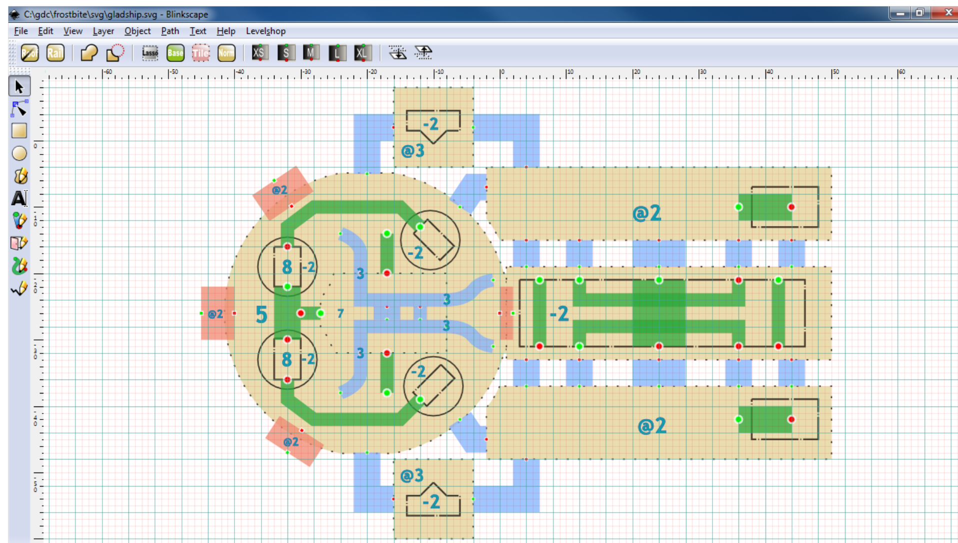


Figure 4.1.1: LevelShop in Action. Beige areas are floors. Red areas are portal connectors; blue areas are floor connectors, and green areas are ramp connectors. Courtesy of Harvey Fong.

Despite the obvious problems with this approach, most video games are developed using this method. It is very rare to see any improvements in the content creation process during game development, as the constraints of having to ship a game within a set time frame minimize the opportunities present for research and development. Consequently, development of accessible tools is an area in which academia can make a major contribution to game development. Universities and other academic working groups can assume risk; failure at the research level at the university does not result in the cancellation or termination of a product.

If we look at what the industry itself has produced, the most promising initial work has been produced by Harvey Fong and his LevelShop research project.[17] Originally started as a secret incubator project at Electronic Arts, Fong developed LevelShop as a rapid prototyping tool.

Inspired by the construction of prototypes in other industries, in particular architectural blueprints, LevelShop was designed to serve as a prototype

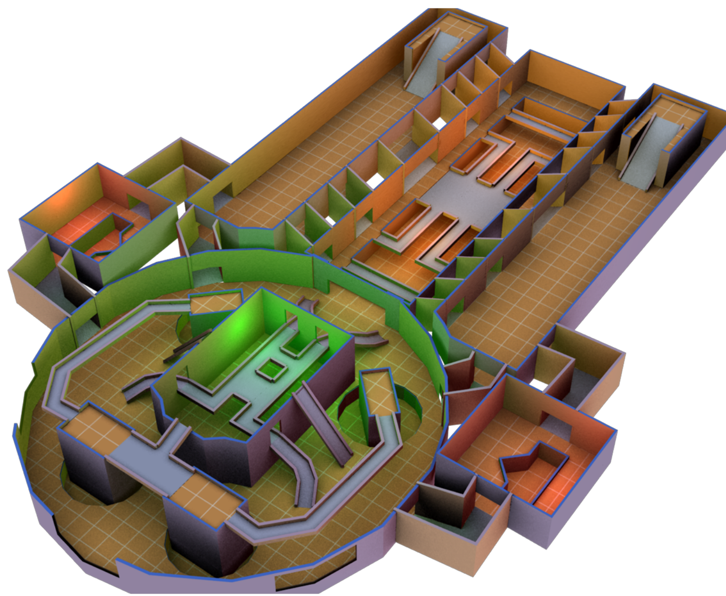


Figure 4.1.2: Output from LevelShop. Note that the geometry is rough and unfinished.

tool for games in the same way that a whiteboard might be used to produce initial sketches for level construction. Fong built LevelShop using a combination of open-source software and off-the-shelf commercial scripting packages. LevelShop itself is based on the open source vector editing program InkScape; unlike traditional level editors, which present the user with three views, LevelShop operates purely in two dimensions. It is worth noting that this reflects our initial design criteria for an improved level editor outlined in Section 2.

The LevelShop workflow consists of three stages.

In the first stage, users draw vector art in a 2D grid-based editor, in the form of a floor plan. The floor plan may be marked with "tags", using the LevelShop text tool, and these tags are understood to modify the geometry produced by LevelShop. A tag of *"4"* indicates, for instance, that this surface is 4 units high. A tag of *"@2"* indicates that the surface starts at two units off of the ground. Tags can be combined; for instance, *"@2;-1"* indicates a surface that starts at a height of two and drops 1 unit below its

starting point, forming a pit. Some surfaces are created with special brushes to produce special effects. Polygonal regions can be connected to each other using “portal” brushes, which produce holes in walls to create doorways and windows. Ramps may also be added to connect two polygons of different heights. Geometry is also allowed to overlap - the 2D draw order of the geometry reflects a “stacked” construction in the generated game level.

In the second stage, the user exports the level from LevelShop. LevelShop creates a SVG (Scaleable Vector Graphics) file, which is run through a Python script and then sent to a server running Houdini, an off-the-shelf 3D rendering and modelling package. In the third stage, Houdini converts the 2D representation of the level into a 3D representation by using standard modelling techniques. When geometry operations are called for, they are performed in three dimensions using traditional CSG based techniques. The resulting output from Houdini is then saved in a file format that can be loaded into a game engine.

In Figure 4.2, I see the output from LevelShop. Note, in particular that areas such as the guard fence surrounding the pit does not have a “clean” border. Rather, there are cracks at the side of the pit, and at multiple other areas in the level where regions collide.

LevelShop has been used to create prototypes for levels in a number of commercially shipped titles, including *DarkSpore*. It is in use as an internal tool throughout multiple divisions of Electronic Arts. Fong’s approach, however, suffers from the fact that he is a technical artist and not a programmer. Worlds produced in LevelShop cannot be used directly in a video game, which is our end goal. In particular, Fong’s tool suffers from his reliance on CSG based methods and lax requirements concerning watertight, robust manifolds in his output geometry. Nonetheless, LevelShop provides two important insights into our problem:

1. Fong, through LevelShop, has defined a collection of components capable of describing the content of a video game level, in broad strokes, across a multitude of video game genres. In fact, his collection of components - doors, ramps, regions, et cetera - can be viewed as an example of an architectural pattern language.

2. Fong presents a basic framework for what I should be looking at in a level editor: a two-dimensional process that takes a marked-up blueprint, possibly layered, and produces a full 3D level.

Furthermore, his work has been designed from the framework of an artist and based on an artist's needs, and has been validated by its successful deployment to produce prototype levels for multiple video games. These results validate his methodology and his design, and indicate that I should consider his work as a valid starting point in the creation of an improved level editor. In fact, LevelShop forms one of the two major inspirations for the work produced in this thesis. Our goals in constructing an improved version of LevelShop, from the perspective of a computer scientist, are as follows:

1. I wish to eliminate Fong's reliance on CSG-based techniques, and on external middleware packages.

2. I need to add support for more complicated and detailed geometry - ideally, I can use the resulting package to produce geometry for the entire level, and not simply for prototypes.

3. I would like to support real time, or nearly instantaneous, updates of the world geometry inside our level editor. (Fong's system involves a lengthy round trip, including send data to an off-site server.)

I will demonstrate such a system in this section.

4.2 Planar Sweep Algorithm

A first attempt to build a better version of Fong's Levelshop uses the idea of a "planar sweep" algorithm from computational geometry. I start at the bottom of the map, and move to the top, building a three-dimensional graph of the extruded map as I go. I then iterate across the graph to produce filled faces.

Fong's original list of criteria geometry is as follows. Polygons in LevelShop are classified as one of the following:

- ROOMS. A room consists of a start or end height. Rooms are the basic geometry of a LevelShop level; they may, or may not, have walls or a roof. By default, they have a floor, but even this can be disabled.[1] Rooms can be nested within rooms; in this case, a room becomes a pit or a raised area, depending on whether it has a positive, or a negative, offset tag.
- FLOOR CONNECTOR. When placed outside of two rooms on a layer, the floor connector forms a connection between two rooms of different heights, in the form of a ramp. (When a floor connector is drawn inside a room, it becomes a walkway from a room to another room, or a walkway from a room to itself.)
- PORTAL CONNECTOR. Essentially, portal connectors cut through walls. They are used to create doors and windows connecting rooms with the rest of the world.
- ROOF CONNECTOR. This should be more accurately named "highest surface connector"; the Roof Connector connects the highest surface of two rooms together in an area.
- WALL CONNECTOR. The purpose of this is unknown, as Fong merely mentions its existance without elaborating on its function. An educated guess would be that it connects two wall units together - for instance, to create a connecting ramp between two parts of a castle.

Levelshop also has the concept of layers; geometry can be moved onto, or off of, layers to create different effects (or just for ease of organization on the part of the level designer.) Every piece of geometry has a start and an end, defined in terms of a height and an offset tag. For instance, a standard room has a height of 4 meters (translated into in-engine units.)

[1] LevelShop uses the "bottomless" tag in order to specify that a room should not have a floor. In effect, this creates a bottomless pit.

As far as I am concerned, I mainly wish to worry about rooms. All of the various connectors simply modify room geometry, and are events that occur on rooms. As Levelshop does, I will make the assumption that no rooms intersect each other - that is, a room is either completely contained in another room, or is completely outside it. (This is to prevent degenerate cases such as pits flowing beneath walls, et cetera.) Any time something happens to a room that affects the wall geometry, I treat it as an intersection event. A portal starts intersecting at a certain Y coordinate, and finishes intersecting at a certain Y coordinate. At each intersection event, I update the shape of the current slice of the room geometry, and then link everything together in the final stage.

The algorithm is as follows:

I include some results produced by our implementation of the LevelShop algorithm. As you can see, it is capable of generating a wide variety of interesting, if box-shaped content.

4.3 Weighted Skeleton Systems

As described, the algorithm in section 4.2 produces perfectly adequate rooms. It is suitable for constructing rough drafts of levels to examine gameplay flow. It cannot, however, produce more detailed geometry with features such as true roofs, overhanging roofs, pillars, columns, doorframes, or non-portal extrusions.

I will tackle the problem of constructing an improved level editor by combining the ideas behind LevelShop with an improved version of a procedural extrusion technique devised by Kelly and Wonka.[27] First, however, I must review some concepts from computational geometry.

The *straight skeleton* is a device well known to computational geometers. It was first outlined by Aichholzer et al.[2, 1], who note in their abstract that “the straight skeleton provides a canonical way of constructing a polygonal roof above a general layout of ground walls.” More formally, consider a closed polygon P (possibly with holes.) Treat the edges of P as a wavefront, moving inwards at a constant rate in a direction that is perpendicular to any edge;

Algorithm 4.1 Planar Sweep implementation of LevelShop.

INPUT: A collection of simple polygons $P = \{P_i\}$ with:

- associated classifications ["ROOM", "PORTAL CONNECTOR", "FLOOR CONNECTOR", "ROOF CONNECTOR", "WALL CONNECTOR"]
- start and end heights
- possibly, associated markup

OUTPUT: One or more polygonal meshes making up the level geometry.

1. Let P_{ROOM} be the collection of all room polygons.
2. For each polygon pair $P_i, P_j \in P_{ROOM}$, determine whether or not the polygons are contained in each other, intersect, or do not intersect. First, determine if their spans of effect overlap; then, if they pass that, do an AABB test; finally, determine if the two polygons intersect or not using the Bentley-Ottmann algorithm.[6] If I encounter any intersecting polygons, HALT. If $P_j \subset P_i$, register P_j as a child of P_i.
3. For each polygon $P_i \in P_{ROOM}$ that is a root polygon of the scene - i.e.a polygon that has no children and is a room:
   (a) Determine which portal polygons Q intersect P_i using the methods listed above.
   (b) For each portal intersecting P_i, push the start and end times of these intersections, as well as the identity of the portal in question, into a priority queue.
   (c) Sweep upwards, popping events off of the priority queue. As I sweep upwards, I track the profile of the room. When an event occurs, I calculate the new shape of the profile of the room, and connect it to the previously swept upwards profile of the room.
   (d) Cap the polygon if it has a roof. More specifically: if any of the child polygons of P_i would intersect with the room - i.e. are forming domes or other interesting structures - I treat them as holes in the polygonal geometry. I then triangulate the resulting non-simple polygon with holes using a modified ear clipping algorithm as described in [45].
   (e) Put a floor on the polygon, if it has a floor. If the polygon has children, they are treated as holes in the floor.
   (f) For each child of P_i, I repeat the process.
   (g) Bevel P_i if necessary.

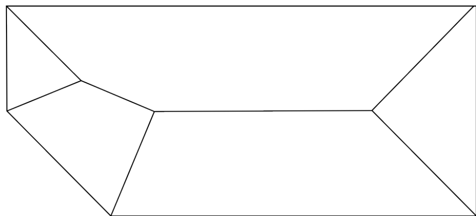


Figure 4.3.1: The straight skeleton of a polygon.

this induces a movement of the vertices of P along the angular bisectors of $P's$ edges. When an edge shrinks to zero, it is allowed it to collapse; continue to do this until P has area zero. The resulting structure left over is known as the *straight skeleton* of P.

Let me make a few observations. First, these straight line segments partition P into a collection of monotone polygons. Second, if I take the straight skeleton and extrude it upwards along the vertical axis, the result forms a convincing and pleasing roof.

Much ado has been made about straight skeletons; unlike their close relative, the medial axis, I cannot construct a straight skeleton in linear time by an incremental algorithm or a divide-and-conquer approach.

A number of algorithms have been proposed to compute the straight skeleton. The best approach to date involves keeping circular lists of active vertices representing closed regions. The basic algorithm simulates the propagation of the wavefront through the straight skeleton, tracking edge events. Edge events are divided into two categories: *edge collapse events* and *split events.* Split events divide a polygon into multiple polygons, and are caused when an edge collides with a non-adjacent edge. Edge collapse events are caused by the length of an edge shrinking to zero, and are caused when two adjacent edges collide.

The wavefront propagation algorithm computes all possible edge events, sorted by time, and places them in a priority queue.[2] The algorithm then iterates through queue events, maintaining the active wavefront, and adding

[2]Strictly speaking, I use a heap that I think of as a priority queue.

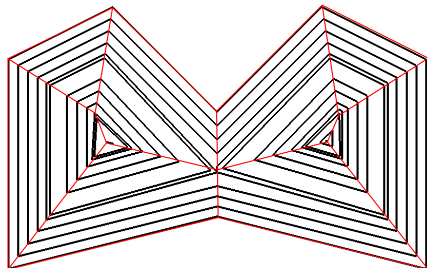


Figure 4.3.2: The straight skeleton of a polygon can be computed via edge collapse and edge split events.

and deleting new collision information after each event as edges appear and disappear. Computing the list of possible events requires the program to determine the intersection of the angular bisectors of each edge, represented as rays. A polygon with n vertices has n angular bisectors, and hence a time and space complexity of $O(n^2)$. This algorithm is due to Felkel and Obdrzalek.[16]

It would be adequate to simply have a robust straight skeleton solver that took "roof" polygons from the algorithm described in section 4.2 and capped them. This would produce convincing roof geometry for buildings, while still allowing for chimneys and similar objects. However, I can do be better by adding one extra degree of freedom to the straight skeleton description.

Kelly's main observation[27] is that we can generalize the straight skeleton to a *weighted straight skeleton* by allowing the wavefronts to move inwards at different rates. By allowing the edges of the wavefront to advance at different speeds - or, in fact, to not advance at all! - we can produce unique and interesting effects. The weighted straight skeleton can also be generalized to include negative weights; this can create effects such as extruded column geometry or Japanese pagodas.

Generalizing the straight skeleton to weighted approaches presents a number of difficulties. For one thing, as Kelly notes, the weighted straight skeleton is no longer uniquely defined. For another, any issues that are present in a regular straight skeleton algorithm with regards to precision - or lack thereof - are made worse by the existence of variable rate wavefront propagation.

To address these issues, Kelly and Wonka refine the algorithm of Felkel and Obdrzalek.

First, Kelly and Wonka introduce a clustering mechanism to the priority queue. When an intersection event occurs, the heap is searched for nearby events within a bounding cylinder. These events are all resolved at the same time, reducing error. This case most commonly occurs when multiple regions meet at a single point, as in the case of an octagonal pagoda.

Second, I have already noted that Felkel and Obdrzalek separate events into two categories: edge collapse events and edge split events. Kelly and Wonka decide that this distinction is unimportant, and instead introduce the notion of a *generalized intersection event*. The bisector approach of Felkel and Obdrzalek is discarded completely; instead, I directly compute the intersection of the swept wave in three dimensions, searching for intersection points. When intersection points are found, they are clustered - first by height, then by distance on the swept plane - and organized into a sequence of chains.[3] The chains are then collapsed until each chain consists of exactly three points and two edges - the operation that corresponds to an edge collapse event - and are then re-oriented, in a manner that corresponds to an edge split event.

Finally, Kelly and Wonka introduce additional classes of events. In order to support lofting by a profile, they allow the user to specify a set of weights associated with heights. I should emphasize that this is not simply building a profile of events; rather, I have heights and angles. Angle-changing events are pushed into the weighted straight skeleton priority queue alongside the intersection events. When an angle change event is hit, it requires a recalculation of all intersection events. Kelly and Wonka also add “profile events”, similar to how our algorithm in Section 4.2 handles portal connectors, “horizontal offset” events to support straight-out extrusions, and “restart” events to support roof geometry with multiple overhangs. The additional events, to their mind, justifies a change of nomenclature; they distinguish the simple “weighted skeleton” without a lofted profile from a “procedural extrusion.”

[3] The original paper does not mention the algorithm used here to collate events; I used dynamic programming.

Algorithm 4.2 Kelly and Wonka's procedural extrusion algorithm.

INPUT: A simple polygon P and a collection of edge profiles E.

OUTPUT: A procedural extrusion.

1. Initialize empty priority queue of events.
2. Determine the intersection of any edges of P based on the current weights by computing the intersection of the appropriate angular bisectors. Push the intersections to the priority queue, ordered by intersection height.
3. Push edge profile weight change events to the priority queue, ordered by height.
4. While the queue is not empty:
   (a) If the current event position is greater than the sweep position, update the sweep position and rebuild the geometry to this point.
   (b) Remove one event from the top of the queue.
   (c) Handle the event that I removed, noting that it may, in turn, add more events to the queue or remove intersections.

By rebuilding the planar sweep algorithm of Section 4.2 to process procedural extrusions using the weighted skeleton method, I end up with a system that is capable of producing complicated geometry. Our final algorithm combines traces of our original algorithm and the Kelly and Wonka algorithm for procedural extrusions.

1. For each polygon pair $P_i, P_j \in P_{ROOM}$, determine whether or not the polygons are contained in each other, intersect, or do not intersect. First, determine if their spans of effect overlap; then, if they pass that, do an AABB test; finally, determine if the two polygons intersect or not using the Bentley-Ottmann algorithm. If I encounter any intersecting polygons, HALT. If $P_j \subset P_i$, register P_j as a child of P_i.

2. If any given P_i and P_j are going to connect to each other during the sweep process, merge them into a new room P_k. To do this, I approximate the expanded size of each room by an axis-aligned bounding cylinder, based on the maximum expansion permitted by any given edge profile in a room. I then test bounding cylinders to see if they intersect, and if so, I merge the room geometry of P_i and P_j into one room P_k (which may possible have multiple chains at the start.)

3. For each polygon $P_i \in P_{ROOM}$ that is a root polygon of the scene - i.e.a polygon that has no children and is a room:

   (a) Add all weight change events from the edge profiles of P_i and all horizontal offset events from the edge profiles of P_i, to the priority queue.

   (b) Determine which portal polygons Q intersect P_i using the methods listed above. For each portal that intersects P_i, push an interrupt event into the priority queue.

   (c) Update all intersections of angular bisectors for P_i based on the starting profile.

   (d) While the priority queue is non-empty:

      i. Pop events z off of the priority queue.

      ii. If $z's$ event position is higher than the current sweep level, update the sweep level (and render out geometry.)

iii. Handle event z:

A. If z is a generalized intersection event, resolve the generalized intersection and update the polygon chains for the sweep as described in Kelly and Wonka.

B. If I encounter a profile offset event or an edge direction event, update the priority queue and edge weightings; recalculate intersection events.

C. If I encounter a portal start event, intersect the currently active chains with the portal. Add an edge to the active chain in the shape of the portal, and flag it as a portal start edge.

D. If I encounter a portal end event, collapse that edge and remove any geometry from consideration that was attached to the extruded portal.

E. If I encounter a profile end event - i.e. I have encountered the end of a profile that does not produce roof geometry - traverse the chain attached the profile end event, looking for other profiles that have ended (or produced profile end events themselves.) Cap the geometry, and remove that section of the chain from consideration (in a manner which is equivalent with collapsing all edges in the chain to a midpoint.)

F. If I encounter an anchor event, process it as per Kelly and Wonka.

(e) Put a floor on the polygon, if it has a floor. If the polygon has children, they are treated as holes in the floor, and are extruded downards or upwards depending on their markup tags.

(f) For each child of P_i, I repeat the process.

(g) Bevel P_i if necessary.

4. Process any polygons that are RAMP CONNECTORS, WALL CONNECTORS, or FLOOR CONNECTORS. (Implementation details are trivial, as this is simply lofting geometry.)

This system lets us produce a number of exciting results, for instance the ones depicted below. All of these scenes were produced by a programmer,

not a trained artist, and should not be interpreted as an artistic statement.

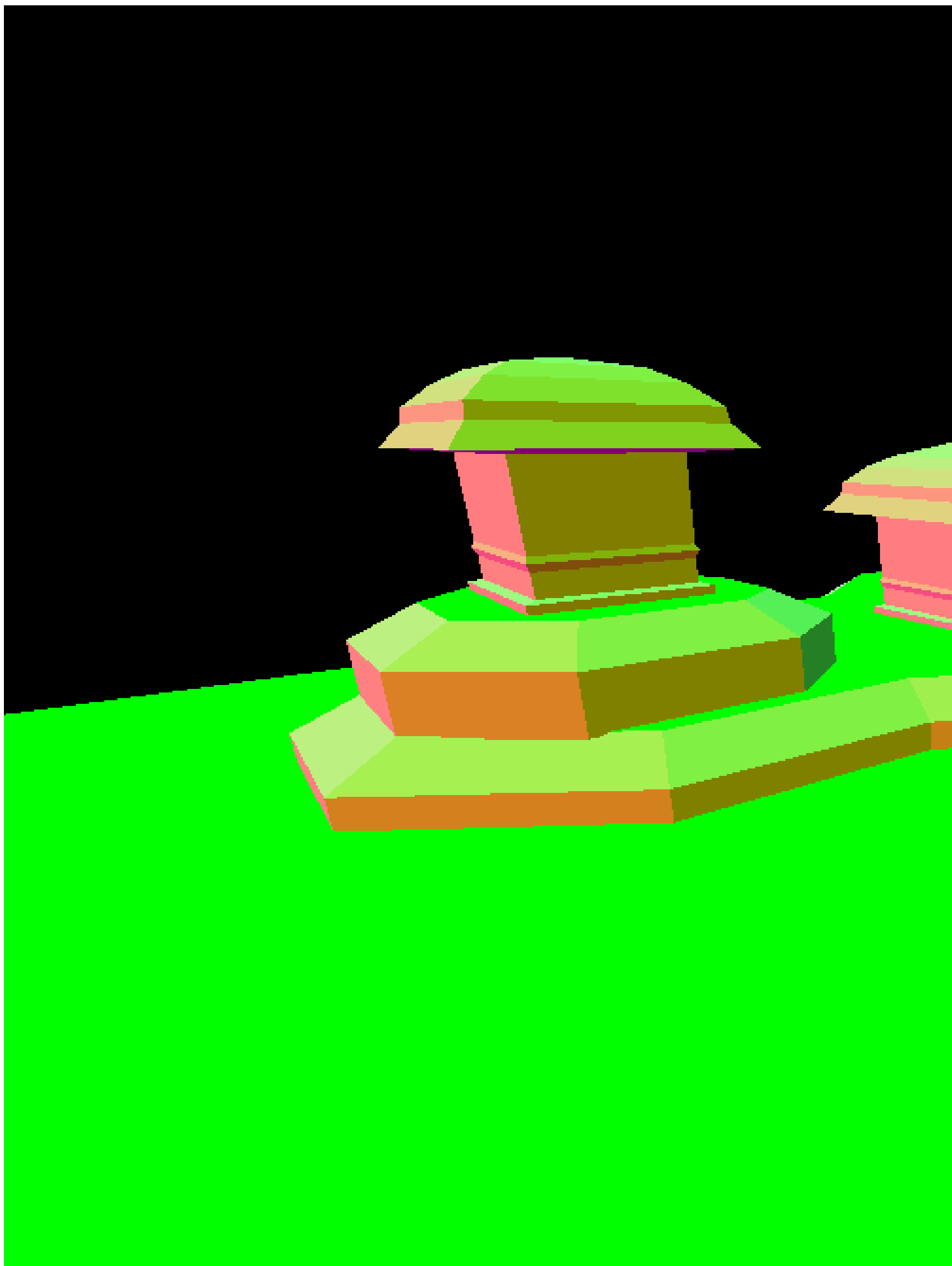
I added a slight nicety to the Kelly and Wonka algorithm that I am suprised was not in the original paper. Procedural extrusions only end when the weighted straight skeleton is reached; I would like to have the ability to create flat roofs as well as bevelled roofs. Accordingly, I let each profile end with a "roof" or a "flat", which I tag in the editor. In the event I reach the end of a profile and a "flat" tag is reached, I walk the chain until I run out of consecutive flat events. I then treat that sub-chain as its own polygon, cap it, and continue extruding the new chain upwards.

4.3.1 Weighted Skeleton Systems and the Motorcycle Graph

The procedural extrusion work of Kelly and Wonka extends the straight skeleton methods of Felkel and Obdrzalek in order to support weights on a skeleton. The authors seem to have favoured the priority queue and planar sweep method for its simplicity, and the elegant integration of an event-based priority queue that they could later bend to their needs.

Eppstein, it should be noted, originally defines the weighted skeleton system in [15], where he lists it as an open area for investigation; the notion is not originally due to Kelly and Wonka. The algorithm they present is a generalization of the ideas originally presented by Felkel and Obdrzalek, and building upon the general consensus in the computational geometry field that a priority queue based, planar sweep algorithm is the way to go. Kelly and Wonka's paper does introduce a number of complicated special cases, and is difficult to follow and implement. Kelly and Wonka also give no guarantee that their algorithm is robust, simply noting that "it did not fail on a large dataset"; finally, they perform no analysis of the average case and worst-case runtimes of their algorithm.[4]

[4]It would seem that the run-time of the algorithm is $O(n^2m)$ where n is the number of vertices in the extrusion, and m is the total number of weighting changes in all profiles being applied to the extrusion. As in the Felkel and Obdrzalek algorithm, we have n angular bisectors, each of which must be tested against each other. However, each time we change the weights at a given height, we must recalculate all the intersections.



Huber and Held present a new approach[25] based on a series of ideas by Eppstein and Erickson[15], later refined by Cheng and Vigneron[10]. They invite us to consider a structure that is parallel to the straight skeleton, the *motorcycle graph*. Given a collection of vertices V on the plane, I place a motorcycle at each vertex with a given direction and (constant) speed. I assume that the motorcycles are delicate; if one motorcycle crosses a path on the plane that has been previously created by another motorcycle, it will crash.[5] Cheng and Vigneron prove:

Theorem 1. *For a simple polygon P, the motorcycle graph $M(P)$ of P covers the reflex arcs of the straight skeleton $S(P)$ of P.[10]*

For a simple polygon in the plane, and the case of the unweighted straight skeleton, Huber and Held achieve great benefits by embedding a motorcycle graph into the waveform construction given by Aichholzer et al. The key advantage of their approach is that, at every stage of the collapse of P, the motorcycle graph $M(G)$ divides P into convex subregions. This reduces the number potential intersection events dramatically.

Both Huber and Held, and Kelly and Wonka, acknowledge that the Felkel-Obdrzalek algorithm seems to suffer from precision and robustness issues when faced with real data. Kelly and Wonka approach the problem by introducing a selection of tolerance and clustering based approaches; however, they provide no guarantees as to the algorithm's efficiency in terms of proof.

The Huber-Held algorithm uses wavefront propagation, similar to the standard approach used by Felkel and Obdrzalek, and Kelly and Wonka. The genius of their approach lies in how they modify the wavefront to create a structure called the *extended wavefront.* In particular, if I have a simple polygon in the plane P, I compute the motorcycle graph $M(G)$ whose vertices are the reflex vertices of P, and where each motorcycle moves with unit speed in the direction of the angular bisector. I then merge P with $M(G)$ to create the extended wavefront, and simulate wavefront propagation in the same manner as the Felkel-Obdrzalek paper.

[5]Another way of visualizing this is to pretend that the motorcycles are, in fact, lightcycles from the movie Tron. A similar setup was employed in this movie as a plot device.

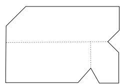


Figure 4.3.4: The Motorcycle Graph subdivides a region into convex subregions. Image after Stefan Huber.

The key insight of Huber and Held is that every topological change in the extended wavefront over the algorithm's lifespan is indicated by the collision of two neighbouring vertices of the extended wavefront. (This fact is also used, implicitly, by Kelly and Wonka in the construction of their generalized intersection events.) They generalized the results of Cheng and Vigneron to the following Theorem:

Theorem 2. *For a simple polygon P, let $M(P)$ be the motorcycle graph of P where each motorcycle created has the same velocity of the corresponding reflex vertex. Then $M(P)$ covers the reflex arcs of the straight skeleton $S(P)$ of P.*

Combining everything known, Huber and Held conclude that if a split event occurs, it must be bounded by a line in the motorcycle graph; hence, a reflex vertex will never move beyond the motorcycle graph trace lines. Additionally, as the motorcycle graph $M(G)$ partitions P into convex sub-regions, during the propagation of the extended wavefront only neighbouring vertices can collide.

If I follow this line of thought to its natural conclusion, Huber and Held show that if I am given the motorcycle graph $M(G)$ of a polygon P, computing the straight skeleton of P takes $O((n+k)\, log\, n)$ time, where n is the number of vertices and k is the number of switch events. This is considerably better than $O(n^2)$. In practice, they show that their approach takes $O(n\, log\, n)$ time, as k turns out to be negligable.

Huber briefly discusses the problem of the weighted straight skeleton in his Ph.D thesis. He notes that it is not clear what the relationship is between the weighted straight skeleton and the motorcycle graph, and that in fact certain key properties of the straight skeleton do not extend to its

weighted counterpart. It is not even clear that, given two lines advancing at different constant speeds along their perpendiculars, the intersection of said lines also advances at a constant speed. In fact, I can show that this is true by simple geometry - which will also let us determine a closed-form formula for the speed of the vertex. Given lines A and B, if I advance both lines along their perpendicular direction by amounts t_A and t_B, the resulting collection of lines forms a parallelogram. The speed at which the vertex advances from one side to the other, during this time, is the length of the diagonal of the parallelograph, and I can compute this via an application of the law of cosines: $c^2 = a^2 + b^2 - 2ab\cos\gamma$.

It is also not even clear in the case of a weighted straight skeleton when a vertex is reflex; what does it mean for an arc to be reflex? Huber gives an example of a vertex which, at first glance, appears to be reflex, but where the surrounding arc has an interior angle less than $\pi/2$. Huber gives a definition for a "reflex" vertex that does not use angles, but rather which is based on angles within the wavefront produced by the edges surrounding the vertex.

I conducted a few experiments with the motorcycle graph, and Huber's definition of reflexivity. There is sufficient evidence to conjecture:

Conjecture 3. *For a bounded, simple polygon with strictly positive vertex weights, the reflex arcs of the weighted straight skeleton are indeed covered by a motorcycle graph, where the velocity of said motorcycle is computed via the speed of the advancing edges connected to the reflex vertex.*

I have been unable to prove this conjecture at this time. Consquently, this line of attack is not useful for our procedural content generation system, but I remain hopeful that it will in the future. It is also not clear at all that this holds with negative or zero vertex weights.

One interesting caveat of the motorcycle graph, however, is that it allows us to resolve the "ambiguity" problem noted in Kelly and Wonka. Kelly and Wonka point out that the weighted straight skeleton is ambiguous: given two edges that meet to form a parallel line, it is possible for them to intersect in a manner that does not admit to a consistent resolution. The motorcycle graph provides us with a dispute resolution mechanism for this situation. First note, this can only happen if a polygon P is concave. Hence P must have a reflex vertex and hence the two edges colliding must be split by a

motorcycle graph in the extended wavefront. But since they are split by the extended wavefront, they are never given an opportunity to collide with each other over time in the extended wavefront - and the ambiguity is inadvertently resolved.

4.3.2 Efficient Parallelization of Level Creation

The relationship between motorcycle graphs and straight skeletons indicates that a parallel algorithm for efficiently computing a straight skeleton may not exist. In particular, I have the following elegant result due to Eppstein and Erickson[15]:

Theorem 4. *MOTORCYCLE-GRAPH is P-COMPLETE under LOGSPACE reductions.*

It is unclear as to whether $NC = P$ or not, but most computer science researchers conjecture that this is not the case. In that case, any $P-COMPLETE$ problem - such as the creation of motorcycle graphs - cannot be efficiently parallelized. As Huber and Held[25] give a polynomial time, log-space reduction from STRAIGHT SKELETON to MOTORCYCLE-GRAPH, I can conclude:

Corollary 5. *STRAIGHT-SKELETON cannot be parallelized efficiently to run in NC-time if $NC \neq P$.*

This result clearly extends to weighted straight skeletons. Parallelization, therefore, must happen at the level of individual components, which are efficiently parallelizable. If I wish to farm work to multiple processors, I can simply split up the room geometry. As per our existing algorithm, I note that any polygons that may touch each other during their extrusion are merged into a common chain. If I select our polygon batches after this merge, I can be assured that they do not touch each other.

4.4 Procedural Extrusion and Model Synthesis

It is not immediately clear that I can apply any of the techniques from our procedural extrusion system to the model synthesis techniques described in Chapter 3. However, procedural extrusion is compatible with Merrell's paper on *continuous* model synthesis.[38]

Merell's extension of the ideas of discrete model synthesis to the continuous case works by assuming the existence of an exemplar that is a simple, closed, polyhedron (or collection of simple, closed, polyhedra.) A synthesized model is created in a way that satisfies a *locality condition*: each point $M(x)$ on the synthesized model M must locally resemble some point $E(y)$ on the exemplar model E.[6] Note that even though models are discrete, the condition itself is a continuous condition - hence, continuous model synthesis.

The actual implementation of continuous model synthesis is much simpler, and can operate in either two or three dimensions. I will focus on the two dimensional case. The exemplar model, in two dimensions, consists of a simple polygon. The slopes of the lines making up the edges of the polygon are computed, and lines with these slopes are extruded in an infinite pattern across the plane. These new lines subdivide the plane into a number of faces; combinations of these faces make up the new shape of the synthesized models.

In the same way as tiles in the discrete form of model synthesis are given a set of possible states, so too are the vertices produced by the intersection of edges on this infinite plane within the bounds of our synthesized model M. A vertex state in M designates all the faces in the closed disc surrounding M as either being full, or empty. A vertex state is said to be *compatible* with the exemplar mesh E if it satisfies the continuous locality criterion.

This leads to the following algorithm:

I can modify the continuous model synthesis algorithm to generate input geometry for procedural extrusion by tagging edges in the exemplar with specific profile IDs, and propagating them throughout the synthesized model.

[6] Of course, this is a very old idea from topology and analysis. Consequently, I can formalize this by using an $\varepsilon - \delta$ style definition: the point x is said to be *locally similar* to y if there exists $\varepsilon > 0$ such that for all vectors δ such that $||\delta|| < \varepsilon$, $M(x + \delta) = E(y + \delta)$

Algorithm 4.3 Continuous Model Synthesis algorithm.
INPUT: Exemplar mesh E.
OUTPUT: Synthesized model M.

1. Create lines parallel to the edges of E to form solid regions, faces, edges, and vertices.
2. Find a list of all acceptable states at each edge and vertex, and store this as the configuration list $C(M)$.
3. While there exists an unassigned edge or vertex:
   (a) Randomly select a new state s from $C(M)$.
   (b) Assign state s to v; add the new vertex and state to the model.
   (c) Remove from $C(M)$ all configurations that are now incompatible with our choice of state s for vertex v.

As a result, I add a new condition to the vertex: a vertex state is compatible with an exemplar in our modified synthesis scheme if it satisfies the locality criterion *and additionally* satisfies the requirement that the configuration of edge profiles must match the configuration of edge profiles at the same point on the exemplar mesh where the locality criterion is satisfied. The resulting output, in two dimensions, can then be passed to the procedural extrusion algorithm.

Naturally, I can also pass generated meshes from the procedural extrusion algorithm to the standard continuous model synthesis algorithm.

Chapter 5

Content Creation for Sparse Virtual Texturing

In a typical interactive 3D application, a virtual world consists of a collection of surfaces (usually with a polygonal representation) and a collection of texture data that is mapped onto these surfaces. In order to minimize space requirements, texture data is typically tiled, repeated, and blended across the world to create the illusion of more texture information being available than is actually present. With storage space now becoming more affordable, this has become an artificial restriction; I can now store sufficient texture information on a computer such that every pixel of rendered output on the screen can have its own unique texel of texture data. This begs the question: how do you efficiently author unique textures that span an entire virtual world? For that matter, how do you efficiently render an eight gigabyte texture when you only have access to under a gigabyte of memory on the GPU?

Sparse virtual texturing[4] addresses the second problem by assuming that the virtual world is uniquely textured using a very large texture (of size $64k \times 64k$ or greater). This texture is then divided into rectangular subregions that are paged in as needed. This introduces new problems: all of the geometry - static or dynamic - in our virtual world must be packed, somehow, into this giant texture. In particular, dynamic geometry cannot be elegantly loaded and displayed without creating a new texture parametrization that includes it. Accordingly, every dynamic element that may potentially occur in our game world - for instance, every possible monster that a player may encounter - must have its texture data unwrapped and packed

into our large texture. Efficient automatic texture atlas generation has been partly solved with the introduction of methods such as signal-specialized parametrization and least-conformal square parametrization; efficient packing of charts, however, is an NP-complete problem (via a reduction from BOX-PACKING)[18]. Producing this texture packing requires the creation of a new texture parametrization every time geometry is changed, added or removed from the virtual world. Accordingly, creating a world with a sparse virtual texture requires a significant investment of art resources and access to a large quantity of computing power, such as a render farm.

An additional problem with sparse virtual texturing is encountered when attempting to implement texture filtering. There are two options for texture filtering: one can either attempt to use the GPU filtering hardware, or one can do perform point sampling and interpolate between these values ourselves. Neither of these methods are perfect; in particular, using the GPU for bilinear filtering with a texture atlas requires padding the edges of cached texture regions with a one page pixel border. As a consequence of discontinuity between chart sampling directions[19], the resulting filtering is not guaranteed to be seam free. Higher-order filtering methods such as trilinear filtering and anisotropic filtering exascerbate this problem; again, a choice must be made between manual sampling or increased padding regions. As the complexity of the filtering to be performed increases, so does the amount of necessary padding needed. 4x anisotropic filtering, for instance, requires 4 extra texels in each direction along a page boundary per mip-map level[8]. Regardless of what solution a developer chooses to implement, a significant investment of technical effort is required to produce artifact-free results, often with a strong cost attached in terms of efficient texture memory usage.

I propose *parametrization-free virtual texturing*, a new virtual texturing method that can solve both of these problems. Our work is a hybrid technique that combines the original concept of virtual texturing - paging texture data to the GPU only when it is visible - with a parametrization-free texturing system that also, conveniently, solves the problem of filtering. By using mesh colors, an alternative texturing construct recently proposed by Yuksel et al.[50], I can efficiently texture large worlds in an on-demand fashion without having to find a global parametrization of the world over the plane, without having to worry about higher-order filtering, and while achieving interactive performance rates.

5.1 Background and Related Work

Parametrization-free virtual texturing is a combination of two ideas: parametrization free texturing schemes and virtual texturing. I have elaborated on some of this material in Section 2, but I will restate it here in more detail.

The first mention of virtual texturing in the graphics literature is in the form of an unpublished technical report by Darwyn Peachey at Pixar. An early form of virtual texturing was implemented for SGI's InfiniteReality project in the form of clipmaps.[46] In its current form, "sparse" virtual texturing was first conceived by John Carmack for the video game "Enemy Territory: Quake Wars", and was first discussed by Sean Barrett at the Game Developer's Conference in 2008.[4] In this form of virtual texturing, a large two dimensional texture is split into a number of square pages. Pages are loaded on demand based on the results of a feedback pass and are packed into a page table that is indexed by a redirection table in the form of a large, two-dimensional texture map. Barrett refers to this form of texturing as "sparse" virtual texturing to distinguish it from other forms of virtual texturing, owing to the fact that the lookup table is sparsely populated and only points to those pages and mip levels that are visible on the screen for any given frame. More recently, Mittring [39] and Van Waveren have given talks ([23]) about their experiences shipping virtual texturing systems in video games. A recent comprehensive survey of virtual texturing can be found in [43].

A number of systems have been proposed for parametrization-free texturing that use octrees as the fundamental construction primitive. TileTrees[29] map 2D textures onto the faces of octree leaves, using existing GPU texture mapping hardware. The tiled nature of TileTrees make them an interesting candidate for virtualization, but they are not compatible with anisotropic filtering or dynamic geometry. The system that has the most in common with our approach is the PTex system developed for Disney by Burley and Lacewell.[7] PTex stores texture mapping information per face, requiring no explicit parametrization pass beyond choosing a level of detail for a Catmull-Clark patch. However, it is not suitable for real-time graphics as it requires access to information on adjacent faces to perform filtering. Since PTex interfaces with a Renderman-style offline rendering system, Burley and Lacewell did not consider how to efficiently render parametrization-free textures in real time, other than packing visible faces into a texture cache style chart

and explicitly uploading packed texture coordinates. This introduces seams and does not permit efficient virtualization of scene primitives.

Mesh colors[50] are a recently developed alternative texture primitive that do not require adjacency information to achieve correct filtering on or near adjacent triangle edges. Instead, they use a barycentric distribution of color data across the surface of a triangle. Because this parametrization is defined on triangle vertices and evently across edge boundaries, color values on or near neighbouring edges are only dependant upon data associated with the triangle on which they are to be computed. In this respect, mesh coloring defines a continuous function over the surface of the colored mesh. Very little research has been done on actually shipping games with parametrization-free content.

5.2 Mesh Colors

5.2.1 Review of Mesh Colors

I will now briefly recap the original mesh colors algorithm as described by Yuksel et al. in [50]. For full details, readers are referred to the original paper.

Mesh colors are defined across a triangular face using barycentric coordinates. Color samples are stored on each vertex, and evenly across the face and edges of a triangle. At their lowest level of detail, mesh colors are equivalent to vertex colors. When rendering a triangle fragment, Yuksel considers the closest three samples to that fragment and then linearly interpolate between them. If R is the resolution of the mesh colors of a triangle, and P is a point on the triangle expressed in barycentric coordinates, the nearest color samples to this point and their respective weighting are determined by computing $i, j, k = \lfloor RP \rfloor$ and the fractional weighting $w = w_i, w_j, w_k = \lfloor RP \rfloor - B$. If $w_i + w_j + w_k = 0$ then I am at the sample point $\lfloor RP \rfloor$ and use the color C_{ij}. Otherwise, if $w_i + w_j + w_k = 1$, I use color samples $C_{ij}, C_{i+1,j}$ and $C_{i,j+1}$ with weighting w_i, w_j, and w_k. If $w_i + w_j + w_k = 2 = 2$ the nearest color values are $C_{i,j+1}$, $C_{i+1,j}$ and $C_{i+1,j+1}$. The weights are then $[1, 1, 1] - w$ respectively. If two triangles share a common edge at the same resolution, and these edges share the same set of colors, continuity is guaranteed across the edge bound-

Figure 5.2.1: Mesh colors are distributed evenly across the surface of a triangle. Left to right: a triangle at $R = 1$ consists only of vertex colors; a triangle at $R = 2$ contains color data at both vertices and edge midpoints; a triangle at resolution $R = 4$ contains vertex, edge and face colors.

ary.

Mesh color mip levels are constructed from the highest level of subdivision by throwing out levels of detail. Trilinear filtering across edge seams is implemented without adjacency information (unlike in the PTex model) by requiring that shared edges have the same color information across that edge. Anisotropic rendering operates in the same way; anisotropy across edge seams is taken care of at the subpixel level by the GPU (assuming the use of multi-sample anti-aliasing.)

5.2.2 Practical Real-Time Implementation

While Yuksel et al.'s original mesh coloring paper states that mesh colors are suitable for both offline and real-time graphics systems, certain elements of the original implementation are not as suitable for run time performance as they could be. Consequently, our implementation of mesh coloring differs from the original version in several regards. I would also like to note some general implementation issues that I encountered while writing a mesh color renderer.

In the original mesh coloring paper, each vertex, edge and face in a mesh is given its own unique index into a color array. When two faces share a common edge, for example, those two edges point to the same area of memory that is shared by both pieces of geometry. This is suitable for off-line rendering where pointer arithmetic and branching are cheap; however, for real-time rendering I am more interested in eliminating branching and in promoting cache efficiency. In our implementation, I chose to duplicate the vertex and edge colors assigned to given faces. If an artist changes a vertex or edge color during run time, I track this change throughout the original mesh using an adjacency structure, updating colors on other faces as needed.

One immediate result of this decision is that it is now possible to predict the number of color samples needed at resolution R. When $R = 1$, a triangle has three colors on it (the three vertex colors.) When $R = 2$, a triangle has six colors on it (the three edge colors, plus the three vertex colors.) In general, for a given resolution R, the triangle has $R + 1$ color samples along each edge, and consequently a total of $(R+1)(R+2)/2$ color samples along each face; this is, of course, the Triangular number formula.[49] Given an array of mesh colors for a triangle at resolution R, color C_{ij} can therefore be found at position $(R-j)(R-j+1)/2+i$ in this array. For practical reasons, I pack these 1D arrays into 2D textures on the GPU and index them using the efficient normalization "gem" presented in [31].

Mip levels are calculated using the GLSL $dFdx()$ and $dFdy()$ functions. I tried computing mip levels using the rate of change of barycentric coordinates as the input, but I found that neighbouring polygons would have mismatched mip levels. Since I use mip levels not only for filtering purposes but to select texture resolution for visible pages, this was unacceptable. This is caused by highly isosceles triangles (or "skinny" triangles, in the vernacular of the original paper); as barycentric coordinates change more rapidly on short edges as opposed to long edges, this would create visual artifacts.[1] Instead, I compute mip levels using the triangle coordinates, in object space, as an input. The resulting mip level is then computed as follows: $mip = 1.0 - clamp(0.5 \cdot log_2(d), 0.0, 1.0)$ where $d = max(dot(dx, dy))$. On most hardware, $dFdx()$ and $dFdy()$ is computed using a 2x2 approximation. I suffer a mild loss of quality as a result when using nearest mip level filtering, but it is not noticeable when using trilinear filtering. As per the original mesh coloring paper, I require mip levels to be powers of 2. Each mip level then corresponds to a decrease of one power of 2 in terms of the actual face resolution.

Some GPU shader compilers do not correctly handle the *floor()* function when attempting to compute the floor of the same, flat color interpolated across all vertices of a triangle. To compensate for this, I use the *flat* keyword introduced in GLSL 1.30 to specify that attributes such as starting page location and face resolution should be preserved without interpolation.

[1] In the original paper, these artefacts are partially alleviated by the inclusion of anisotropic filtering.

This successfully resolved the problem, and is added as a per-graphics card workaround. Additionally, the weighting calculations to determine what combination of points to use require small adjustments for floating point tolerance.

Because barycentric coordinates are not available on the GPU, for each triangle in our mesh I pass the barycentric coordinates of the triangle's three vertices as three-dimensional texture coordinates. Correct barycentric coordinates for a fragment are computed by linear interpolation in the fragment shader.

5.2.3 Virtual Paging for Mesh Colors

Our algorithm simulates virtual paging on the GPU. Each triangle in the scene is identified with a unique 32-bit integer (conveniently corresponding to an RGBA texture with 8 bits per channel.)[2] The top 24 bits are taken as a page number. The final eight bits represent the order of a triangle with respect to a given page. Each page consists of an array containing $256 \cdot (R+1)(R+2)/2$ colors, where R is the maximum resolution of the mesh colors for any given triangle in that page. Polygons can be distributed amongst pages as needed. For instance, some quantity of page space can be reserved for world geometry, and some portion of page space can then be dynamically allocated for entities. Note that this introduces an endian dependency into our code which must be accounted for when authoring content that must run on both big-endian and little-endian hardware.

Similar to a traditional SVT renderer, I first perform an identification pass to determine which mesh colors need to be loaded. I write the results of the identification pass into a small "feedback pass" framebuffer object, occupying 1/16th of the total space used by the frame buffer. The RGB channels of the feedback buffer store the 24-bit page ID; the bottom 8 bits are used to store the computed mip level. When a given page is seen, I load the entire page; this prevents us having to store the bottom 8 bits of ID data in the

[2] This introduces a theoretical limit to the number of polygons that can be textured in the world at once - namely, this system cannot handle more than 2^{32} triangles worth of data. As the world space coordinates for this amount of geometry alone would occupy 48 gigabytes worth of data, I am not too worried about the ramifications of this limitation any time soon!

framebuffer, and conveniently allows us to use them to store our desired mip level. This data is then read back from the GPU to the CPU using pixel buffer objects for efficient, non-stalling, DMA transfer across the bus. In order to prevent stalls due to waiting on the GPU, I allow for some lag during the feedback pass. In our implementation, the results of the feedback query for a given frame are used up to 4 frames later.

For each page that is visible, I determine the maximum mip level of resolution of any triangles that belong to it. If a given triangle is visible, I determine which page it lies on and the maximum mip level needed to display it. If the data is not loaded, I begin loading it in another thread. By default, each polygon in the scene that is visible has a set of colors stored for each vertex that corresponds to the lowest mip level; I can then fall back to vertex coloring for distant objects, or those objects that do not have mip levels calculated.

Paged data is loaded into two $8k \times 8k$ textures, each with 4 levels of mipmapping. The first $8k \times 8k$ texture is used to store "high resolution" pages which contain faces that have a maximum per-face resolution of 256 or more. The second $8k \times 8k$ texture stores "low resolution" pages that are used when displaying data that, at the current mip resolution, has a per-face resolution of 16 or less. (These sizes are, of course, adjustable according to the user's specifications.) I originally wanted to use a system of equally-sized texture layers, but I found that modern hardware simply does not have enough on-board texture memory for this to be feasible. For mip levels where I cannot fit an entire page into a $4k \times 4k$ table (any resolution above $R = 2^8 = 256$), I split pages across multiple layers if necessary; one 512×512 page is mapped to two 256×256 pages.

I then build a $4k \times 4k$ texture that serves as a page table, and perform a second rendering pass to render the actual colored geometry. Our motivation for using an indirection table stored as a texture, instead of manually redirecting triangles to the correct page and location on the CPU, is that this allows us to upload vertex buffers to the GPU and then to ignore them. This is one of the main advantages of regular sparse virtual texturing, and I seek to preserve it. The top 12 and middle 12 bits of the face ID are sent as texture coordinates to the GPU, along with the location in the page and the desired face resolution. The shader then determines the actual location

of the triangle's page in the texture array. If a face has a higher desired resolution than the resolution currently loaded, I simply drop down to a lower resolution. The rest of the algorithm proceeds as defined in the original mesh colors paper, including the relevant material on trilinear and higher order filtering. Trilinear filtering requires a total of nine texture reads per fragment. Bilinear filtering requires a total of four texture reads per fragment.

Like other virtual texturing implementations with latency and background loading, I can suffer page faults if I attempt to render texture data that is not loaded in time. Since GPUs do not have the ability to handle page faults, if a page is not loaded at the desired mip resolution I fall back to the highest resolution that is loaded in memory. In the event that no mip resolutions are loaded in our texture array for a page that I wish to render, I fall back to mip level $R = 1$ which corresponds to linear interpolation of vertex colors. To ensure that vertex colors are always available, I pass them to our shader - as vertex colors.

5.2.4 Mesh Preprocessing

As observed in the original paper by Yuksel et al., traditional mesh coloring suffers from visual artifacts on isosceles, or "skinny", triangles. For character models, this is not necessarily an issue as skinny triangles will likely end up occupying relatively little screen size. For a virtual world, skinny isosceles triangles can occupy considerable real estate both on screen and in the world. Additionally, it may be advantageous to break large triangles, requiring a high resolution mesh coloring, into smaller triangles that can survive with a lower resolution mesh coloring. This frees up valuable page space in the higher-resolution page table.

To solve both of these problems, I add an optional remeshing stage to our world loader. For each triangle in the scene, I overlay a set of points in the two-dimensional space spanned by the triangle's tangent plane. First, I subdivide each edge of the triangle into regular intervals, similar to the subdivision used by the mesh coloring algorithm. Next, I overlay a grid of points across the triangle's tangent plane, of a density chosen by the user. I then select only those grid points that satisfy two criteria: first, they must be contained inside the triangle, and second, they must not be within a certain,

user-defined distance of any triangle edge. After the point list has been generated, a standard Delaunay triangulation is performed (with respect to the two-dimensional vector space spanned by the basis vectors of the triangle's tangent plane.)

Points created along the edges serve to subdivide the triangle edges and to ensure that the convex hull of the point set is equal to the hull of the original triangle. Because I perform the generation of edge points in a way that is consistent along shared edges, triangles can be tessellated locally but can be guaranteed to satisfy certain global properties. The resulting mesh has the same surface shape as the original mesh, and is guaranteed to be free of T-junctions. Furthermore, triangles produced by the Delaunay triangulation are guaranteed to be approximately regular, in the sense that the minimum angle of any angle in the triangle is minimized. This makes the resulting output ideal for mesh coloring. This algorithm is also embarrassingly parallel, in that the computation can be divided evenly amongst any number of processors without interdependencies. I stress that this tessellation is only sometimes necessary when adapting input to our system; content authored with mesh colors in mind does not need any treatment.

5.3 Results

This picture shows our parametrization-free virtual texturing system in action. Here, I painted a landscape consisting of approximately 200,000 triangles using my stamp-based detailing system. If run at the highest level of detail, the mesh can be illustrated with a mesh color resolution of 256 colors on each edge; this produces a total of 1.7 *terabytes* of uncompressed coloring information. Despite this, my system is able to paint on the landscape in real time, with bilinear filtering and mip-mapping enabled.[3] The visualization of the landscape is limited purely by the amount of available video memory and maximum texture size supported by the hardware.

With just seven brushes, I can produce a nice landscape consisting of a mixture of mountainous and grassy textures, pieces of roadwork, and flowers.

[3]While my system does implement full trilinear, it is currently disabled due to a hardware bug on my graphics hardware at the time of writing.

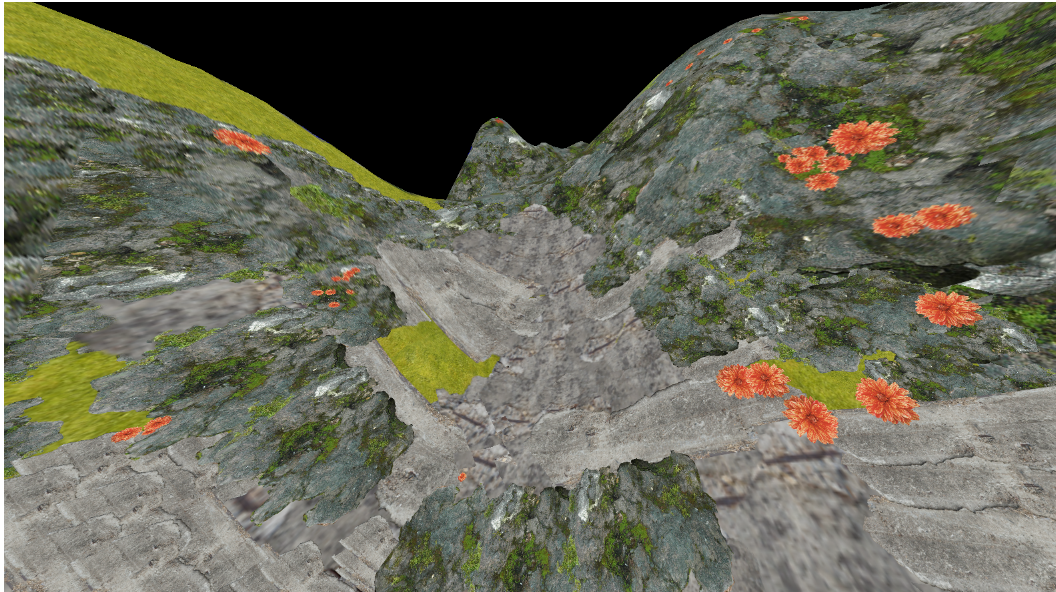


Figure 5.3.1: Parametrization-free virtual texturing in action.

Better work could, of course, be produced by a professional artist; as always, this work is not intended to be an artistic statement.

It is worth noting that the mesh color system implicitly adds some stretch at low mip levels to the terrain - in particular, on the sides of the valley. This is more apparent at lower resolutions, and can largely be resolved by oversampling.

5.4 Baking Parametrization-Free Virtual Textures

It is not always desireable to use parametrization-free virtual textures in an engine. Without a truly programmable graphics pipeline, such as the proposed Larrabee architecture, mesh color based approaches and their ilk will always underperform their fixed-pipeline compatriots. Furthermore, mesh coloring eliminates any possibility of efficient cache usage on a standard GPU; texture caches are normally read assuming that the fragment pipeline will be wanting elements nearby in a rectangular region, not a contiguous line in

memory. Accordingly, while it is important that parametrization-free virtual textures should be (and are!) fast enough on modern graphics cards for real time preview, I should be able to bake out our texture to a continuous block of memory.

In doing so, I wish the following criterion:

1. No mapping should be performed by the artist. (Otherwise, what is the point?) Rather, all coordinates should be assigned by the algorithm.
2. The mapping algorithm should take into account a fixed texture budget for the level, and that budget should be independent of the actual mesh coloring signal present on the world.
3. The mapping algorithm should allocate more space to those regions of space where a signal is relatively high-frequency, or cannot be efficiently reconstructed by bilinear interpolation. It should allocate less space to areas of low frequency signal.

Fortunately, an algorithm already exists to do this. In a series of two papers, Sander et al. propose a method that takes as input a signal on a mesh, parametrized across the vertices of a triangle, or across a subdivision scheme of a triangle, and converts it into a texture. Their approach, *signal-specialized parametrization*, considers the problem of signal mapping to a texture as a general map from a domain D in two dimensions across to a surface S in three dimensions. They introduce a signal Q mapped on S by a function $g : \mathbb{R}^3 \rightarrow \mathbb{R}^n$ and attempt to find a good surface parametrization of S - in the form of a map $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ - by considering how well the composite map $h = g \circ f$ from D to Q is approximated when resampled. A signal specialized metric representing the integrated squared error of the parametrization over the surface, with respect to the signal Q, is given as an integrated metric tensor.

Fortunately for us, there is a relationship between the integrated mesh tensor and the subdivision scheme employed when computing a mesh coloring. By default, Sander et al. consider the problem of an integrated metric tensor being computed for a triangle with per-vertex attributes that are linearly interpolated. It can be shown that, for this case, the metric tensor for the triangle T with respect to the signal is a constant at any point on the triangle. Integrating the metric tensor is then as simple as multiplying the

constant value of the metric tensor by the area of the triangle T in space S.

To handle a signal that has more detail than can simply be described at the vertices, Sander et al. subdivide the triangle into smaller sub-triangles, using the same subvision scheme as Yuksel et al. Computing the integrated metric tensor then turns into converting the integrated metric tensor at each small triangular subregion - where things are, again, constant - and then summing the result.

Baking mesh colors using signal-specialized parametrization is a two-fold process. First, I divide our textured level into patches that are topologically equivalent to the unit disc. Second, I expand each of these parametrizations to fit into a known region using signal-specialized parametrization with piecewise linear construction, and the coarse-to-fine chart reconstruction algorithm outlined in Sander et al. I can then subdivide our textured level into patches that are topologically equivalent to the unit disc[4] and expand each of these parametrizations using Sander et al.'s coarse-to-fine optimization method.[44] I then pack the resulting patches - inflated to be squares - in a large, sparse virtual texture using a "Tetris Packing" algorithm.[34]

In practice, most game developers do not bother implementing Sander et al.'s algorithm; a great deal of math and programming is required, and it is not easy to implement. Fortunately, a reference implementation of the algorithm is included with the DirectX 9 SDK, and it is compatible with our mesh coloring approach. The normal problem with Sander's algorithm is figuring out what to actually pass as a metric tensor, but here what is meant by a "signal" and how to convert it into the form required by the algorithm is blessedly obvious.

[4]For instance, by cutting each disconnected component mesh in the level along the minimum spanning tree. I will revisit this sort of cutting algorithm in Chapter 6.

Chapter 6

Applications of Barycentric Coordinates

A major problem with the parametrization-free texturing algorithm presented in Section 6 is that current graphics hardware does not allow access to per-face data or barycentric coordinates in the fragment shader. The naive, yet traditional, solution to this problem - as suggested by Yuksel[50] and others - has been to send per-face data to the graphics card by requiring each face to have three unique vertices. This is incredibly inefficient; if I have n triangles, I must send $3n$ vertices to the graphics card. It is therefore desireable to find a way to work around this limitation.

In this section, I introduce three methods that work around this limitation, allowing access to both per-face data and to non-vertex data in a fragment program. All our methods are based on ideas from combinatorics. Our first method is a more refined, yet still naive, approach that allows for the submission of general per-face attributes. Our second method allows access to barycentric coordinates on a mesh from within a fragment shader; our third method then refines the second technique to the problem of storing per-face data. Additional applications will be given.

6.1 General Per-Face Data

Our first approach takes advantage of a language quirk in the OpenGL shading language.

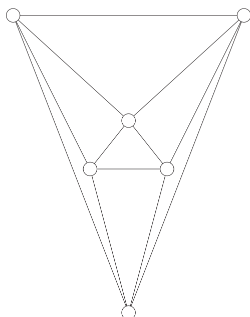


Figure 6.1.1: The triangular graph T_4 does not have a bipartite matching between its vertices and faces.

In the OpenGL shading language, a vertex attribute can be specified as "flat" in the fragment shader. Flat vertex attributes are passed through to the fragment shader without linear interpolation. In this case, the ordering of the vertices on each face of the mesh is used to determine which of the triangle's vertices is the "provoking" vertex. The flat attribute then receives its value from the provoking vertex's attribute data. In most cases, I cannot use this to embed per-face information into the vertices of a mesh. For instance, consider the triangular graph T_4. Despite being both planar, connected, and a topological disc, it has seven faces and only six vertices. I am therefore motivated to answer the question: given a collection of vertices and faces, when can I uniquely assign provoking vertices to each face in a mesh?

I address this problem by converting it into a bipartite matching problem. Given a mesh with per-face data, I construct a bipartite graph $G = \{V_1, V_2\}$ as follows: V_1 is the complete set of vertices of the mesh, and V_2 is the set of faces of the mesh. A vertex $v \in V_1$ and a face $f \in V_2$ are joined by an edge in G if and only if v belongs to f . Thus the problem of uniquely assigning provoking vertices to each face of the mesh reduces to the problem of finding a complete bipartite matching on V_2. This matching does not always exist; a necessary and sufficient condition for the existence of such a matching is given by Hall's Theorem.[22] To rectify this, I find the maximum bipartite matching on the graph (for instance, using a max-flow algorithm) and assign a unique provoking vertex to each face using the resulting bipartite matching. For any faces that have not yet received a unique provoking vertex, I

duplicate one vertex belonging to that face and use it as the provoking vertex.

Because the provoking vertex depends on a given vertex being first in the draw order, this approach eliminates our ability to submit data to the GPU in the form of triangle strips. I can, however, still optimize for vertex cache locality.

This technique has multiple issues. First, there is no guarantee that the video card is not tripling the vertices to achieve this effect on hardware. A cursory inspection of the register documentation for the AMD R600 graphics card chipset indicates that the "flat" attribute is supported in hardware.[3] It is unknown whether this is supported in hardware on other GPU parts or not. Second, as noted above, I can no longer send face data to the GPU as triangle strips.

I will return to this problem in section 7.3, after I have developed enhanced combinatorial techniques to deal with it.

6.2 Barycentric Coordinates in the Fragment Shader

Without loss of generality, assume that I am interested in a watertight, non-self-intersecting, connected, triangulated mesh - i.e. a standard triangulated model of the form used in computer graphics.

For a fixed triangle T with vertices u, v, and w, any point p on the triangle can be expressed as a linear combination of the form $p = p_1 u + p_2 v + p_3 w$ such that $p_1 + p_2 + p_3 = 1$.[12] Taken as a vector, the weights $[p_1, p_2, p_3]$ determine the barycentric coordinates of p. This parametrization of space is often used for efficient computation in computer graphics; for instance, barycentric coordinates play important roles in high performance texture synthesis and higher-order interpolation of vertex attributes across triangle faces.

I wish to determine the barycentric coordinates of a point on a triangle in a fragment shader. Despite the fact that - for well-behaved meshes - any given fragment only truly belongs to one triangle, modern graphics

hardware does not support per-face information. The naive, yet traditional, solution to this problem has been to duplicate the vertices of a mesh face; for each face that a given vertex belongs to, I create one copy of the vertex imbued with face-specific parameters. This is inefficient; in addition to increasing the number of vertices I send to the graphics card, it also thwarts the graphics card's ability to cache vertices, and the programmer's ability to submit data to the GPU in the form of triangle strips. For algorithms that must be applied to entire scenes, this approach clearly duplicates a lot of data, and introduces inefficiencies in terms of storage capacity. On hardware supporting geometry shaders, I can produce barycentric coordinates as per-primitive output after the vertex shader executes; however, geometry shaders are not supported on all platforms. Additionally, the speed implications of this approach are unknown, and it would be desireable to sidestep this process altogether. Therefore, I must convert a per-face piece of information (an ordering of barycentric coordinates) into a per-vertex piece of information that is compatible with modern graphics hardware.

First note that while barycentric coordinates are a per-face assignment, the barycentric coordinates of the vertices of a triangle T are always $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. What I am really searching for is a way of assigning barycentric coordinates to vertices on the mesh such that, if some vertex v has barycentric coordinates $(1, 0, 0)$ (say) with respect to any one triangle on the mesh, it has the same barycentric coordinate for all the triangles that it belongs to. In general, this is not possible on all meshes. From this, I can infer that I can avoid duplicating most vertices in a mesh, but I may have to duplicate some of them. Since this is an improvement over the naive solution, I accept it. Since there are only three possible barycentric coordinate assignments for a triangle vertex, I see that this is equivalent to finding a 3-coloring of the vertices of the mesh. 3-vertex coloring has been studied exhaustively, and in general determining whether or not a given graph is 3-colorable is an NP-complete problem. Fortunately for us, a sufficient condition exists that shows when I can color the vertices of a planar triangulation of a polygon that is homeomorphic to the topological disc:

Fact. Any triangulation of the topological disc in which all interior vertices have even degree is 3-colorable.[40]

I have moved this proof to the appendix.

This suggests the following strategy:

1. Topologically cut the target mesh in order to make it homeomorphic to a disc.

2. Assure that all vertices in the topological disc interior have even degree.

3. Find a 3-coloring of the disc, and use this coloring to assign barycentric coordinates correctly to vertices.

4. The resulting barycentric assignment can then be passed to the vertex shader as vertex information; the linearly interpolated values can then be used to determine the barycentric coordinates of a fragment with respect to its parent triangle in the fragment shader.

Methods for efficiently computing topological cuts are already described in the literature - see, for instance, [21]. I do not propose to discuss these algorithms here. In order to ensure that all vertices in the generated topological disc interior have even degree, I select vertices that have odd degree and find the shortest path between that vertex and any boundary vertex using Djikstra's algorithm.[11] Traversing an edge has a cost of zero if it is an edge connecting two nodes of odd degree, and one otherwise; this encourages efficient cutting. I then cut the disc along that path. This increases the degree of the vertex by one, and I do not need to worry about changing the degree of any other nodes; any nodes affected by this cut are now duplicated and placed on the boundary. Interior vertices of the new, cut disc are then tested for odd degree against the new boundary. I repeat this process until all odd nodes are eliminated.

To find a 3-coloring of the disc once all interior vertices of the graph have even degree, it is not sufficient to implement a simple greedy algorithm. In graph coloring problems, the greedy algorithm for graph colors is typically very sensitive to vertex ordering, and it is fairly easy to construct a triangulation of the unit disc where the greedy algorithm fails to color it without adding a new color or additional vertex splits - both of which are unnecessary. Since I know that the graph is 3-colorable, I know that a solution exists; I could therefore consider a backtracking search for the 3-coloring. This is still too inefficient for our needs; I would ideally like to operate in linear time.

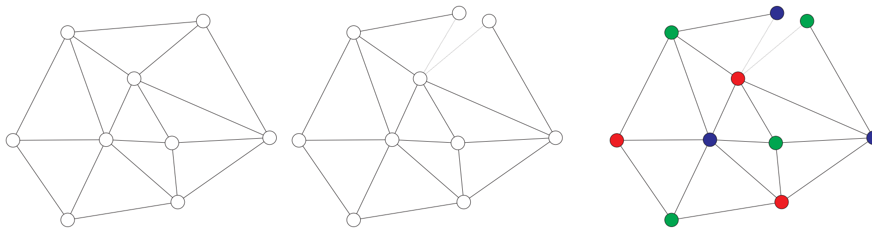


Figure 6.2.1: Left to right: a) A disc to be colored, containing an interior node with odd degree. b) I apply our cutting algorithm in order to make this node even degree. c) The resulting disc can then be 3-colored using a simple, greedy algorithm.

Algorithm 6.1 3-coloring a triangulation of the unit disc.

1. Choose a triangle T; color it black, and color its vertices *red*, *green*, and *blue* in clockwise order.

2. For each edge of the triangle T, look at its neighbours S_i. Since each S_i shares an edge with T, it has two colors assigned. Color the triangle S_i white, and color the vertices of S_i *red, green, and blue* in counter-clockwise order.

3. Repeat until the disc is colored. If I encounter any triangles that are not colored black or white, they are "bow tie" triangles which share only one common vertex with another triangle; I can color them black or white, as long as I honor the one vertex that is colored.

I therefore propose the following algorithm, based on the construction of the 3-coloring in the existence proof that I give in Appendix A.

This algorithm can be implemented either recursively, or with a stack.

The resulting coloring can then be converted into triangle strips, and optimized for vertex cache locality. If the mesh already has a set of texture coordinates, either artist generated or automatically assigned, the algorithm already has a decomposition of the mesh into charts that are equivalent to the unit disc. In that case, the cutting and coloring algorithm is applied once, for each chart, to efficiently re-use the pre-existing cuts in the manifold.

The restriction that our disc interior may only contain vertices of even degree is not necessarily as problematic as one might think. In several common cases - for instance, in the case of a grid-based terrain mesh - I can enforce the requirement that all interior vertices must have even degree by construction. In these cases, I can bypass the search for vertices of odd degree in the disc interior. In the case of artist-generated meshes, meshes generated with edge loop modelling tend to produce vertices of degree four, with vertices of odd degree appearing only at the start and end of a loop. Hence our technique also works well for organic models.

6.3 General, Per-Face Data, Second Approach

I can now combine our barycentric approach with the method outlined in [35]. Malan gives a description of a technique developed for real-time image quilting on an arbitrary surface, extending the ideas behind "image quilting" as outlined in Section 3 to the problem of texturing an arbitrary surface. In his talk at SIGGRAPH 2011, he outlines a method for storing per-face data, given the existence of barycentric coordinates. Here, barycentric coordinates are supplied using a method somewhere between the method I outline in Section 7.2 and the naive approach of vertex duplication.

Malan's approach to per-face data is as follows. Suppose I wish to associate the float s with a given triangle. I compute a 3-coloring of our mesh, and for each face, I create a duplicate of the red vertex only. I then store the two-dimensional vector $(s, 1)$ in the red vertices, and the vector $(0, 0)$ is associated with all green and blue vertices. During fragment shading, the value in the first component of the interpolated vector will be equal to ks, where $k \in [0, 1]$, and the value in the second component of the interpolated vector will be k. Determining the value at s in the shader then reduces to dividing the first component of the interpolated vector by the second. Compared to the normal approach of duplicating each vertex on a triangle for each face, I only need to duplicate one third of the vertices of each triangle. The other two vertices can safely be shared, and can then be processed by the vertex cache. The resulting structure can also be processed as triangle strips.

The key insight that I present here is that I can produce a more efficient 3-coloring of the mesh vertices than Malan's approach. Malan uses a greedy algorithm for graph coloring, attempting to minimize vertex splits.[1] I can use the same topological methods of section 6.2 to produce a 3-coloring of a mesh; this can then be to store per-face data using Malan's method.

6.4 Applications

As a simple application of our two techniques working in unison, I consider the problem of higher-order normal interpolation. When determining the normal of a fragment, linear interpolation and renormalization of vertex normals is only correct if the surface to which the triangle belongs is flat, like a table. For fragments belonging to curved surfaces, higher-order normal interpolation is desireable. Until now, this has only been possible with the aid of geometry shaders. In a standard implementation of higher-order normal interpolation, a triangle is passed into a geometry shader, whcih outputs three new, distinct vertices containing a) per-face data, distributed over all three vertices, consisting of a set of "edge normals"; and b) a set of barycentric coordinates assigned to each vertex. This information is then used to interpolate vertex normals across the surface of the triangle using a weighted, higher-order interpolation scheme.

Using our system, I can avoid the necessity of a geometry shader and of face duplication. Three edge normals are passed in as per-face data using the method in Section 7.1 or 7.3, ordered according to the winding associated with the barycentric coloring constructed in Section 7.2. The resulting information is then processed in the fragment shader as in the case of the standard approach. What is notable about this approach to higher-order normal inter-

[1] In general, greedy algorithms do not produce optimal graph colorings. For instance, consider the crown graph C_{2n} consisting of the sets of vertices $I = \{i_1, ..., i_n\}$ and $J = \{j_{1}, ..., j_n\}$, and where an edge connects the vertices i_a and j_b whenever $a \neq b$. Since C_{2n} is bipartite, it can be colored with two colors; however, if the vertices are presented in order $i_1, j_1, ..., i_n, j_n$ to a greedy algorithm, the algorithm will color the graph with n colors. However, this is a different class of greedy algorithm from the one presented in [35]; here, the greedy algorithm responds to a lack of an available color by attempting to modify the mesh geometry. Consequently, it is hard to draw conclusions about the efficiency of these graph coloring algorithms from the available literature. This is an interesting open area of research.

polation is that it is explicitly stated by Microsoft as being "a new feature in DirectX 10" that can only be produced with newer, more advanced graphics cards - and I have shown that you can produce the same result on DirectX 9-class hardware. This makes higher-order normal interpolation accessible on consoles such as the Playstation 3 and XBOX 360 that do not support Geometry Shaders, as well as on embedded platforms such as OpenGL ES.

As mentioned previously, our barycentric coordinate packing system can also be used to efficiently render mesh colors and procedural image quilting on meshes. In terms of my thesis, this is where it excels: procedural image quilting over a mesh provides a new tool for an artist seeking to decorate a landscape or a video game level. Accelerating this process in a controlled manner - and making it effectively free for any well-behaved mesh of quads - means that image quilting is now suitable for deployment on any current generation console and video game engine.

Chapter 7

Conclusion

Our goal with this thesis has been two-fold. First, I wish to illustrate that it is possible to create better tools for video game level creation with the current, existing slate of technologies, and that these tools are accessible to game studios today.

Have I created these tools? I feel that the answer is a resounding Yes. This paper presents a number of scenes created rapidly and hastily using our prototyping tools, yet achieving quality that would have taken a professional artist more time to produce. The process by which these scenes has been created was pleasant and inspirational; at no point was there a feeling of restriction. Adding a new building in our modelling package was as simple as saying, "Let it be so!"

Our modelling and texturing tools could easily be used on a next-generation AAA video game, or as the foundation for an independent game developer's masterpiece. The modeling tools presented in Chapter 4 could even be used as part of the video game themselves, in a manner similar to how advanced character creation technology is used as the center piece of gameplay in the game *Spore.*[24] The texturing tools presented in Chapters 6 and 7 are designed with an existing problem in mind: developing a practical toolkit for the creation of uniquely textured worlds. This is an active problem facing the next generation of video game creators, and it remains to be seen if unique textures across entire worlds are practical or a technological dead end. Proper, sophisticated tools will play a role in making this determination.

There are two potential avenues for future work in this area. The first avenue involves extending the techniques presented here. Fong's vocabulary for level architecture could be extended into a true pattern language, and additional features could be added to our planar sweep model to support additional artist requests. I had hoped to add support for more interesting decorative elements in a level, such as destruction and damage created by Voronoi-based methods, but time did not permit this. I could also combine our mesh coloring approach for parametrization-free virtual texturing with automatic assignment models. It is also interesting to consider better reconstruction methods for mesh colors when converted to unique massive textures. For instance, I do not attempt to optimize for any sort of rendering cache locality when packing elements. It is also possible that there are better tools for applying colors to mesh colors than simple linear sampling; a new candidate is a paper by Kavan, Bargteil and Sloan that addresses the problem of reconstructing a signal using vertex colors and a least-squares approximation scheme.[26] Integrating this paper with mesh colors to produce better results is a personal high priority.

The second avenue of work involves identifying new problems in level construction and rendering that I have not identified. Examples of these problems may include decorating levels with props, such as crates, in a unique and interesting way. I am also interested in methods for constructing characters to populate our new levels without artistic involvement, or in producing levels with multi-stage gameplay objectives that are generated randomly. Identifying the problems in this domain is almost as hard as coming up with solutions for them.

Level generation remains a pressing challenge for the video game industry. The industry itself seems powerless to make the games that consumers demand without being dominated by the increased costs associated with advances in technology. This presents a novel opportunity for collaboration between industry and academia: the academy is not tied to any particular video game project, and can investigate blue-sky technologies that will aid in the development of multiple titles across multiple platforms. The industry, meanwhile, can provide a steady source of real-world problems, and the resources to see these problems understood and conquered.

Bibliography

[1] Oswin Aichholzer and Franz Aurenhammer. Straight skeletons for general polygonal figures in the plane. pages 117–126. Springer-Verlag, 1996.

[2] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, dec 1995.

[3] AMD. R6xx family instruction set architecture.

[4] Sean Barrett. Sparse virtual textures. http://silverspaceship.com/src/svt/.

[5] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[6] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28:643–647, September 1979.

[7] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008*, pages 1155–1164, 2008.

[8] Matthaus G. Chadjas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre. Virtual texture mapping 101. In *GPU Pro*. AK Peters, 2010.

[9] Matthäus G. Chajdas, Sylvain Lefebvre, and Marc Stamminger. Assisted texture assignment. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM Press, 2010.

[10] Siu-Wing Cheng and Antoine Vigneron. Motorcycle graphs and straight skeletons. *Algorithmica*, 47:159–182, February 2007.

[11] T.H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[12] H.S.M. Coxeter. *Introduction to geometry*. Wiley classics library. Wiley, 1969.

[13] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, August 2001.

[14] Christian Eisenacher, Chuck Tappan, Brent Burley, Daniel Teece, and Arthur Shek. Example-based texture synthesis on disney's *tangled*. In *SIGGRAPH Talks*, 2010.

[15] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *Proceedings of the fourteenth annual symposium on Computational geometry*, SCG '98, pages 58–67, New York, NY, USA, 1998. ACM.

[16] Petr Felkel and Stepán Obdrzálek. Straight skeleton implementation. In *Proceedings of Spring Conference on Computer Graphics*, pages 210–218, 1998.

[17] Harvey Fong. Levelshop: From grid paper to playable, whiteboxes on demand. Game Developer's Conference, 2011.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[19] Francisco González and Gustavo Patow. Continuity mapping for multi-chart textures. *ACM Trans. Graph.*, 28(5):1–8, 2009.

[20] Austin Grossman, editor. *Postmortems from Game Developer: Insights from the Developers of Unreal Tournament, Black and White, Age of Empires, and Other Top-Selling Games*. CMP Books, 2003.

[21] X.D. Gu and S.T. Yau. *Computational conformal geometry.* Advanced lectures in mathematics. International Press, 2008.

[22] Frank Harary. *Graph theory.* Addison-Wesley series in mathematics. Perseus Books, 1994.

[23] Evan Hart and J.M.P van Waveren. Using virtual texturing to handle massive texture data. NVIDIA GPU Technology Conference, September 21, 2010.

[24] Chris Hecker, Bernd Raabe, Ryan W. Enslow, John DeWeese, Jordan Maynard, and Kees van Prooijen. Real-time motion retargeting to highly varied user-created morphologies. *ACM Trans. Graph.*, 27:27:1–27:11, August 2008.

[25] Stefan Huber and Martin Held. Theoretical and practical results on straight skeletons of planar straight-line graphs. In *Proceedings of the 27th annual ACM symposium on Computational geometry*, SoCG '11, pages 171–178, New York, NY, USA, 2011. ACM.

[26] Ladislav Kavan, Adam W. Bargteil, and Peter-Pike Sloan. Least squares vertex baking. *Comput. Graph. Forum (Proceedings of EGSR 2011)*, 30(4), 2011.

[27] Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph.*, 30:14:1–14:15, April 2011.

[28] David Kushner. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture.*

[29] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 25–31, New York, NY, USA, 2007. ACM.

[30] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Trans. Graph.*, 25:541–548, July 2006.

[31] A. Lefohn, J. Kniss, and J. Owens. Implementing efficient parallel data structures on gpus. In *GPU Gems 2*. Addison Wesley, 2005.

[32] Eric Lengyel. Transition cells for dynamic multiresolution marching cubes. *Journal of Graphics Tools*, 2011.

[33] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, August 1987.

[34] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillo t. Least squares conformal maps for automatic texture atlas generation. In ACM, editor, *ACM SIGGRAPH conference proceedings*, Jul 2002.

[35] Hugh Malan. Real-time image quilting: Arbitrary material blends, invisible seams, and no repeats. In *SIGGRAPH '11: ACM SIGGRAPH 2011 classes*, 2011.

[36] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Chc++: Coherent hierarchical culling revisited, April 2008.

[37] Paul Merrell. Example-based model synthesis. In *In I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 105–112. ACM Press, 2007.

[38] Paul Merrell and Dinesh Manocha. Continuous model synthesis. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 158:1–158:7, New York, NY, USA, 2008. ACM.

[39] Martin Mittring and Crytek GmbH. Advanced virtual texture topics. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 23–51, New York, NY, USA, 2008. ACM.

[40] Bojan Mohar. Some topological methods in graph coloring theory. *Electronic Notes in Discrete Mathematics*, 5(0):231 – 234, 2000. <ce:title>6th International Conference on Graph Theory</ce:title>.

[41] Chris Morris. The next generation of gaming consoles.

[42] Bruce F. Naylor, John Amanatides, and William C. Thibault. Merging bsp trees yields polyhedral set operations. In *SIGGRAPH*, pages 115–124, 1990.

[43] Andreas Neu. Virtual texturing. Rheinisch-Westfalische Technische Hochschule Aachen Bachelor's Thesis, 2010. Undergraduate Thesis.

[44] Pedro V. Sander, Steven J. Gortler, John Snyder, and Hugues Hoppe. Signal-specialized parametrization. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 87–98, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[45] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[46] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.

[47] Geetika Tewari, John Snyder, Pedro V. Sander, Steven J. Gortler, and Hugues Hoppe. Signal-specialized parameterization for piecewise linear reconstruction. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '04, pages 55–64, New York, NY, USA, 2004. ACM.

[48] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH*, pages 153–162, 1987.

[49] Eric W. Weisstein. Triangular numbers. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/TriangularNumber.html.

[50] Cem Yuksel, John Keyser, and Donald H. House. Mesh colors. *ACM Trans. Graph.*, 29(2):1–11, 2010.

Appendix A

Triangulations of the Disc with all Interior Vertices of Even Degree are 3-Colorable

The following is used in section 7.2.

Theorem. *Any triangulation of the topological disc in which all interior vertices have even degree is 3-colorable.*

This is stated, without citation, in Mohar[40], but no proof is given; it is stated as a "well-known proposition." Interestingly enough, the well-known proposition is not cited in the literature, nor is it actually used as a homework problem in any of the major texts on graph theory. Consequently, I include a proof here for completeness's sake.

Recall the following:

Definition 6. Let G be a planar graph. The *dual graph* $D(G)$ is the graph that has one vertex for each face of G, and where vertices x and y in $D(G)$ are connected by an edge e iff the associated faces X and Y in G share a common edge.

Lemma 7. *Let G be a planar graph. Then G is bipartite iff $D(G)$ is Eulerian.*

Proof. If G is bipartite and planar, each face has an even number of edges (otherwise the graph would contain an odd cycle, which is impossible in a bipartite planar graph.) Hence every vertex of $D(G)$ has an even degree. Hence $D(G)$ is Eulerian. Conversely, if $D(G)$ is Eulerian, $D(G)$ must have no vertices of odd degree; hence every cycle of G is an even cycle and G must be bipartite. □

Lemma 8. *A planar graph G is 2-face-colorable iff it is Eulerian.*

Proof. If G is Eulerian, by the previous proposition its dual graph $D(G)$ is bipartite. Since $D(G)$ is bipartite, it is 2-colorable; the 2-coloring of $D(G)$ induces a 2-face coloring of G. Similarily, if G is 2-face-colorable, then $D(G)$ is 2-colorable; ergo, $D(G)$ is bipartite, and by the previous lemma G is Eulerian.

□

I now have everything I need to develop the proof of the main assertion.

Theorem. *Any triangulation of the topological disc in which all interior vertices have even degree is 3-colorable.*

Proof. Let G be a triangulation of the topological disc with all interior vertices of even degree. Since G is a triangulation of the topological disc, it is simply connected; since it has all interior vertices of even degree, it is planar. This implies that G is Eulerian; hence it has a 2-face coloring and all triangles of G can be colored black and white such that no two adjacent triangles share the same coloring. I now color every white face with the colors *red*, *green*, and *blue*, such that these colors appear in clockwise order. Note that this induces a partial coloring of the black triangles: if any black triangle is surrounded by white triangles, then it is colored with the colors *red*, *green,* and *blue* in anti-clockwise order. (See Figure.)

I then extend this coloring to every black face, by applying the colors *red, green,* and *blue* to every black face such that these colors appear in anti-clockwise order.

This is clearly a 3-coloring of G; for if any two vertices r_1 and r_2 share the same color, they must be connected by a common edge belonging to a white or black triangle. Each edge of a triangle contains two vertices of different colors, and so a contradiction ensues.

□

A computer scientist with a background in graphics or computer vision will recognize the basic technique of splitting a triangulated mesh into white and black triangles with different clockwise and anti-clockwise orderings as being similar to the *winged-edge tree* proposed by Baumgart.[5]

Appendix B

Shader Code

B.1 Parametrization-Free Virtual Texturing Shader

For reference, I have included GLSL source code for the two OpenGL shaders used by the parametrization-free virtual texturing system. First, the page shader:

```
[Vertex shader]
#version 130
#pragma STDGL invariant(all)
flat out vec4 face;
flat out float fR;
out vec3 barycentric;

void main()
{
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        barycentric = gl_Vertex.xyz;
        face = gl_Color;
        fR = 8.0;
}

[Fragment shader]
#version 130
#pragma STDGL invariant(all)
```

```
#define saturate(x) clamp(x,0.0,1.0)
flat in vec4 face;
flat in float fR;
in vec3 barycentric;
#extension GL_ARB_draw_buffers : enable

void main()
{
        gl_FragData[0] = face;
        vec3 dx = dFdx(barycentric);
        vec3 dy = dFdy(barycentric);
        float d = max(dot(dx, dx), dot(dy, dy));
        float mip_bias = 0.0;
        float mip = 1.0 - saturate(log2(sqrt(d)));
        float mipLevel = ceil(fR * (1.0 - mip)) / fR;
        gl_FragData[0].x = mipLevel;
}
```

Finally, here is the source code for the actual mesh coloring shader.

```
[Vertex shader]
#version 130
#pragma STDGL invariant(all)
out vec3 barycentric;
flat out vec4 start;
out vec3 vertexColors;
out vec3 pos;
flat out float fR;
flat out float fRPower;
void main()
{
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        start = gl_MultiTexCoord0;
        vertexColors = gl_Color.xyz;
        barycentric = gl_MultiTexCoord1.xyz;
        pos = gl_Position.xyz;
        fR = 8.0;
        fRPower = 256.0;
```

```
        // Lighting vectors

        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_TexCoord[1] = gl_MultiTexCoord1;
        gl_TexCoord[2] = gl_MultiTexCoord2;
        gl_TexCoord[3] = gl_MultiTexCoord3;
}

[Fragment shader]
#version 130
#pragma STDGL invariant(all)

in vec3 barycentric;
in vec3 vertexColors;

#extension GL_ARB_draw_buffers : enable
#extension GL_ARB_shader_texture_lod : enable
#extension GL_EXT_texture_array : enable

uniform sampler2D baseTex;
uniform sampler2D addressTex;
uniform sampler2D lowTex;

vec4 conv_consts = vec4(
        1.0 / 8192.0,
        1.0 / 4096.0,
        1.0 / 2048.0,
        1.0 / 1024.0 );

float CONV_CONST;
vec4 widths = vec4( 8192.0, 4096.0, 2048.0, 1024.0 );
float width;
vec4 skip_mipped = vec4(33153.0, 8385.0, 2145.0, 561.0);
vec4 rows_mipped = vec4(1037.0, 525.0, 269.0, 141.0);
vec4 skip_array = vec4(153.0, 45.0, 15.0, 6.0);
vec4 rows_array = vec4(5.0, 3.0, 2.0, 2.0);
float fudge_const = 0.0;
```

```
uniform float texScale;
uniform float mipBias;

#define saturate(x) clamp(x,0.0,1.0)
#define lerp mix

flat in float fR;
flat in float frPower;
flat in vec4 start;
vec2 lookup;
in vec3 pos;
float page;

vec2 addrTranslation ( float addr )
{
        vec2 TexCoord;
        // texCoord needs to be in centre of texel
        TexCoord.x = mod(addr, width) + 0.5;
        TexCoord.y = floor(addr / width) + 0.5;
        return TexCoord * CONV_CONST;
}

float skip;
vec2 texCoord ( float i, float j, float faceResolution )
{
        float gaussformula = i + ((faceResolution - j)
                * (faceResolution - j + 1.0) * 0.5);
        vec2 startpos = vec2(0, lookup.y / width);
        return addrTranslation(gaussformula + (start.x * skip))
                + startpos;
}

float mipLevel;

void do_fudge()
{
        if (mipLevel < 1)
        {
```

```
                skip = skip_mipped[0];
                page *= rows_mipped[0];
                CONV_CONST = conv_consts[0];
                width = widths[0];
        }
        else if (mipLevel < 2)
        {
                skip = skip_mipped[1];
                page *= rows_mipped[1];
                CONV_CONST = conv_consts[1];
                width = widths[1];
        }
        else if (mipLevel < 3)
        {
                skip = skip_mipped[2];
                page *= rows_mipped[2];
                CONV_CONST = conv_consts[2];
                width = widths[2];
        }
        else
        {
                skip = skip_mipped[3];
                page *= rows_mipped[3];
                CONV_CONST = conv_consts[3];
                width = widths[3];
        }

        lookup.x = 0.0;
        lookup.y = (page);
}

void do_fudge_small()
{
        if (mipLevel < 1)
        {
                skip = skip_array[0];
                page *= rows_array[0];
                CONV_CONST = conv_consts[0];
```

```
                width = widths[0];
        }
        else if (mipLevel < 2)
        {
                skip = skip_array[1];
                page *= rows_array[1];
                CONV_CONST = conv_consts[1];
                width = widths[1];
        }
        else if (mipLevel < 3)
        {
                skip = skip_array[2];
                page *= rows_array[2];
                CONV_CONST = conv_consts[2];
                width = widths[2];
        }
        else
        {
                skip = skip_array[3];
                page *= rows_array[3];
                CONV_CONST = conv_consts[3];
                width = widths[3];
        }

        lookup.x = 0.0;         lookup.y = (page);
}

void main()
{
        vec3 dx = dFdx(pos.xyz * 4.0);
        vec3 dy = dFdy(pos.xyz * 4.0);
        float d = max(dot(dx, dx), dot(dy, dy));
    float mip = 1.0 - saturate(log2(sqrt(d)) + mipBias);

        // get # from 0 to 127

        vec4 pageData = texture2D(addressTex, vec2(start.zw));
        page = pageData.x * 255.0;
```

```
        mip = min(mip, pageData.w);
        if (mip == 0)
        {
                // just use vertex colors

                gl_FragData[0] = vec4(vertexColors,1);
        }
        else
        {
                // mip level from 0 to 8
                mipLevel = ceil(fR * (1.0 - mip));

                vec4 cij;
                vec4 cipj;
                vec4 cijp;
                vec4 cipjp;
                vec4 baseTexel;
                float faceResolution;
                vec3 integerPortion;
                vec3 floatPortion;

                faceResolution = pow(2.0, floor(fR * mip));
                integerPortion = floor(barycentric * faceResolution);
                floatPortion = (barycentric * faceResolution)
                                        - integerPortion;

                if (mipLevel >= 4)
                {
                        mipLevel -= 4;
                        do_fudge_small();
                        cij = textureLod(lowTex,
texCoord(integerPortion.x, integerPortion.y, faceResolution),
mipLevel);
                        cipj = textureLod(lowTex,
texCoord(integerPortion.x + 1, integerPortion.y, faceResolution),
mipLevel);
                        cijp = textureLod(lowTex,
```

```
texCoord(integerPortion.x, integerPortion.y + 1, faceResolution),
mipLevel);
                        cipjp = textureLod(lowTex,
texCoord(integerPortion.x + 1, integerPortion.y + 1, faceResolution),
mipLevel);
                }
                else
                {
                        do_fudge();
                        cij = textureLod(baseTex,
texCoord(integerPortion.x, integerPortion.y, faceResolution),
mipLevel);
                        cipj = textureLod(baseTex,
texCoord(integerPortion.x + 1, integerPortion.y, faceResolution),
mipLevel);
                        cijp = textureLod(baseTex,
texCoord(integerPortion.x, integerPortion.y + 1, faceResolution),
mipLevel);
                        cipjp = textureLod(baseTex,
texCoord(integerPortion.x + 1, integerPortion.y + 1, faceResolution),
mipLevel);
                }

                if (floatPortion.x + floatPortion.y + floatPortion.z
                        < 0.01)
                {
                        // w = 0: sample texture coordinate directly
                        baseTexel = cij;
                }
                else if (floatPortion.x + floatPortion.y
                + floatPortion.z > 1.01)
                {
                        vec3 weights = vec3(1,1,1)
                                - saturate(floatPortion);
                        baseTexel = (cipjp * weights.z)
                                        + (cijp * weights.x)
                                        + (cipj) * weights.y;
                }
```

```
                else
                {
                        baseTexel = (cij * floatPortion.z)
                                      + (cipj * floatPortion.x)
                                      + (cijp * floatPortion.y);
                }
                gl_FragData[0] = baseTexel;
        }
}
```