Automatic Tessellation of Images to Produce Seamless Texture Tiles

UC Berkeley CS-194-26: Final Project

Alex Liu
14 December 2015
Introduction

Textures are one of the most important building blocks of computer graphics. In video games and in computer animation, nearly everything you see on screen is textured in some way or another. Textures are crucial to making 3D geometry appear realistic, and they can add great detail to a scene without much additional rendering. Some textures are made for 3D models, while others are designed to cover large surfaces by seamlessly tiling an image. My final project deals with the construction of the latter type of texture.

Objective

The goal for my final project is to take pictures of patterned surfaces and create tileable texture patches. This should be done programmatically, resulting in a tile that when repeated, appears to be a seamless image that can be applied to 3D geometry. It is also important that the tiled image is perceptively identical to the source image. The algorithm should be efficient in run-time and require minimal user input. Specifically, the algorithm should meet the following:

- Be robust to minor lighting variations
- Easily tile stochastic patterns
- Be able to easily tile most patterns that are a mixture of stochastic and structured
- Tile strongly structured patterns with the aid of manual adjustments
- Always be able to create seamless tiles, potentially losing some of the source’s characteristics

Ultimately, I wish to test the results from this algorithm by texturing a map in Valve’s Source engine (specifically, in CS:GO)
Algorithm

The algorithm has three primary stages

1. User aided extraction of target region from source image
2. Removal of lighting artifacts
3. Tessellation of edges

Extraction

This is the simplest step in the process. Using a GUI, the user selects a region from the source image and provides a desired aspect ratio. The aspect ratio is important because in the Source engine, all textures must have dimensions in powers of two (e.g. 512x512, 1024x2048).

![Image 1. Using the GUI, the user is able to select a specific region within an image](image1.png)

Lighting

The next step is removing lighting effects. Oftentimes when taking pictures of flat planes, the difference in brightness of any two parts of the plane can be quite different, even if the plane is uniformly textured. This can be a result from direct light sources placed near the subject, from diffuse bounce lighting from a nearby object, or from shadows cast by other objects in the scene. In our algorithm, we will focus on reducing the effects of direct light sources and from diffuse light sources. Accurately and reliably detecting shadows is an issue that could warrant its own project.

We approach this problem by assuming that any two large regions of an image should be relatively similar in terms of texture content, and thus any differences in brightness between those regions are attributable to lighting.
To remove lighting effects, I first tried a naïve approach of blurring the image and using the resulting Gaussian to estimate the lighting at a particular point. Using this approximate light map, I then attempted to normalize the lighting across the image.
As can be seen here, using the Gaussian filter to estimate the illumination at each point of the image results in poor balancing. This is due to the fact that the Gaussian filter’s behavior on the edge of the patch is to either use reflected or constant values. As a result, if illumination is consistently falling off towards one side of an image, a simple Gaussian filter will pretend that the trend stops at the image edge, which is likely not true. As a result, a Gaussian driven light balancing algorithm will fare well for the central area of the image, but will fail near the edges, which is the most important area of the image for tessellation!

Since we need to be able to extrapolate how the lighting of an image behaves just outside the boundaries of the image, we need some other way to approximate the lighting map. For this, I turned to b-splines for their ability to smoothly interpolate (and extrapolate, to some degree) a surface using a small number of control points.

*Image 7, 8. Above: another example with carpet. Right: 3x3 tiling of relit sample with \( \sigma = 50 \)*

*Figure 3. An example b-spline showing smooth interpolation using a mesh of control points*
To calculate the “height” of the control points, I split up the image patch into subpatches – each one corresponding to a control point. The average intensity of each subpatch would be the height of the control point. Using this mesh, I was able to produce a smooth light map that more accurately accounted for behavior near the edges of my image patch.

![Image 9. Sketch showing arrangement of subpatches (blue) and control points (orange)](image)

Evidently, using this method produces far better results than using the Gauss estimation. I eventually settled on using a linear basis rather than using a cubic or quadratic basis because the latter two resulted in dramatic curves towards the edges of the image since there are no control points outside the image patch.

Conveniently, using a control mesh lends itself nicely to accounting for diffuse bounce lighting. The above images are from the cork board on the wall (image 2), the reddish lighting on the right side of the patch is a result of the orange-painted wall just to the right of the board (not in frame, but you can see it on the left side of the board as well). All we need to do is perform the light remapping process 3 times. One for each color channel, red, green, and blue.

It makes sense that using a small number of control points to approximate the lighting on a flat surface works well. Such lighting is low frequency in nature. In fact, solving for the intensity of light on a diffuse texture lit by a single source gives approximately the following equation: \( I = \frac{L \cos(\theta)}{r^2} \). Both \( \theta \) and \( r \) change rather slowly across a flat plane (provided the light is at a decent distance away), so the illumination can be closely matched by a few lines. So long as there only a few light sources, the spline control point approach serves its purpose quite well.
Tessellation

Taking the image and making it tileable can be tricky. Textures that are relatively unstructured and noisy, such as concrete or carpet, are pretty easy to tessellate. However, strongly structured textures like brick and floor tile can be difficult. I experimented with several approaches for tessellation, some with better results than others.

The first method I took was the simplest: simply take the image patch and blend it with itself twice – once in the left/right direction, and again in the top/down direction. I used a multiresolution blend instead of a basic alpha ramp to give the algorithm some tolerance to high frequency structure. As a side note, this process makes the patch somewhat smaller, but preserves the aspect ratio.

*Image 11, 12, 13. Carpet Blending. From left to right: Original target patch, resulting tileable patch, 3x3 grid of target patch*

*Image 14, 15, 16. Concrete Blending Same ordering as above*
Blending fails however, once strong structure is introduced. Running it on a brick texture fails in making a seamless tile.

To try and deal with structure, I implemented quilting through lowest energy seam carving. To get my tessellatable tile, I first carve in the left/right direction, and then in the top/bottom direction. The energy function used was simply the absolute difference between the opposite edges of the image. Once I found the lowest seam, I used the seam as the boundary for my blending mask and did multiresolution blending again. The approach is quite similar to the multiresolution blending, except instead of using a straight edge blending mask, I used the least noticeable seam.
The result using seam blending looks much better. Although there still are still areas where the tile transitions awkwardly, it is much less obvious than with straight line blending.

One difficulty with the seam blending was dealing with the corners of the patch. The corners of a tile need to consider all other corners of the tile, rather than just the opposite side. As a result, since blending and seam blending performs two passes in the top/down and left/right directions, the two blends might not agree with each other at the corners.

For example, here is the seam blending’s attempt at tessellating a checkerboard pattern.

![Image 22, 23, 24. From top to left to right:
Source patch,
2x2 tiling result from seam blending,
highlighting seam at corner meeting point](image)

Since the seams for each pass conflict with each other near the corner of the tile, we are left with this ugly artifact. To fix this, we add strong negative energy weights in some clever places to prevent this from happening. By placing negative weights in the marked areas, we can ‘encourage’ the left/right and top/bottom seams to avoid each other.

![Figure 4. Negative weights in blue, example paths in yellow.](image)
After adding these weights, we get a much better result. The seam no longer gets abruptly cut.

Another approach I attempted was to use the Efros-Leung algorithm to do a fill-in-the-blank tessellation. The basic approach was to take the target region and arrange it in a 2x2 grid with a small gap between the tiles.

This however, did not end up working because the algorithm proved too computationally expensive (or I severely messed up the implementation). The setup shown in image 27 would have taken several hours or days to run to completion. Even with a pyramid based SSD search, the algorithm still would have taken far too long to complete at a resolution high enough to be useful in a game engine.
My implementation was by no means very efficient, but even with decent optimizations, it likely still would have been too computationally expensive. In my research, I came across a paper titled *Exemplar-Based Texture Synthesis: the Efros–Leung Algorithm* [1], which described a PCA accelerated implementation of the Efros-Leung algorithm which runs in “acceptable time” [1]. However, the algorithm seemed too complex for me to implement given the scope of my project.

The final approach I tried was reflection. This was motivated by my goal of being able to create a seamless texture regardless of what input was given. Although in most cases, the overall character of the input texture is lost, the resulting texture has no visible seams. In some cases, it can produce unexpected and quite interesting results. I won’t prove it here, but it should be intuitive that reflecting an image across the x and y axes results in an image that can be seamlessly tiled.

As you can see, we get a nice, seamless texture. However, perceptively, it looks quite different from the source patch. This is in part due to the fact that reflecting across the x and y axes results in an image that is radially symmetric around the origin. This catches the eye and can give the image an almost crystalline personality.

For the most part, the reflection approach is suboptimal. However, it is an effective last resort when we are presented with a texture that is seemingly untessellatable. To show this, I drew a random pattern.
This is a pattern that seems to make no sense. In fact, it would be rather difficult for a user to manually tessellate this patch. It is in these situations that our reflection scheme produces superior results.

This concludes the last of the tessellation schemes that I tried. To recall, the approaches I used were:

1. Simple Horizontal/Vertical Multiresolution Blending
2. Lowest Energy Seam Blending
3. X/Y Reflection

(along with the Efros-Leung algorithm, which I unfortunately could not get to work)

Overall, I judge the lowest energy seam blending to be the best approach, with X/Y reflection being a good last resort.
Conclusion

In my project, I attempted to take images and turn them into tessellatable tiles. I produced an algorithm that was dealt well with most textures, but had difficulty dealing with strongly structured textures. In addition, I was able to account for minor lighting inconsistencies within the input image.

I’ll end my report with a compilation of various input patches and what I judged to be the best resulting tessellation from my three tessellation schemes. On the left is the input patch. On the right is the 3x3 tiling from the best algorithm for the job.

_Carpet_. Best result: Seam Blending

_Cork Board_. Best result: Seam Blending
Brick. Best result: Seam Blending

Checkers. Best result: Seam Blending

Night sky. Best result: Seam Blending
Mural.
Best result: Reflection

Wood.
Best result: Seam Blending

Marble.
Best result: Seam Blending
Metal.
Best result: Seam Blending

Some Random Lines.
Best result: Reflection

Concrete.
Best result: Seam Blending
To see these textures in action, I created a map in Hammer (map editor and compiler for the Source Engine)

My textures in action in the Hammer editor

All textures used in this map were created by my algorithm except for the skybox, water, glass, and 3D prefab models (light fixtures and paintings)

Here is the video
References


All pictures were taken by me or taken from Wikipedia and marked with free use.

Those taken or drawn by me are the following (and subsequent processed images):

*Carpet, Cork Board, Metal, Mural, Some Random Lines, Image 1, Figure 4.*

All others are from Wikipedia under fair use.

Figure 1 - [https://upload.wikimedia.org/wikipedia/commons/0/04/UVMapping.png](https://upload.wikimedia.org/wikipedia/commons/0/04/UVMapping.png)

Figure 2 - [https://upload.wikimedia.org/wikipedia/commons/c/c5/Tiled_brick_texture.jpg](https://upload.wikimedia.org/wikipedia/commons/c/c5/Tiled_brick_texture.jpg)

Figure 3 - [https://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Surface_modelling.svg...](https://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Surface_modelling.svg...)

Concrete - [https://upload.wikimedia.org/wikipedia/commons/5/52/Concrete_texture.jpg](https://upload.wikimedia.org/wikipedia/commons/5/52/Concrete_texture.jpg)


Marble - [https://upload.wikimedia.org/wikipedia/commons/2/20/Surface_exterior_vertical_wall...](https://upload.wikimedia.org/wikipedia/commons/2/20/Surface_exterior_vertical_wall...)

Chess - [https://upload.wikimedia.org/wikipedia/commons/d/db/Chess.board.fabric.png](https://upload.wikimedia.org/wikipedia/commons/d/db/Chess.board.fabric.png)