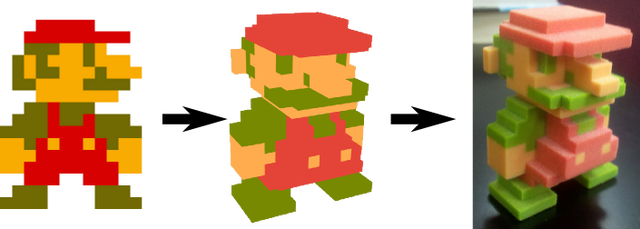
[**Turning 8-Bit Sprites into Printable 3D Models**](https://0fps.net/2012/09/18/turning-8-bit-sprites-into-printable-3d-models/)

Posted on [September 18, 2012](https://0fps.net/2012/09/18/turning-8-bit-sprites-into-printable-3d-models/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

Continuing in the recreational spirit of this blog, this week I want to write about something purely fun.  In particular, I’m going to tell you how to convert retro 16 x 16 sprites into 3D miniatures:

[](https://0fps.files.wordpress.com/2012/09/process.png)

Turning an 8-bit sprite into a 3D model

This project was motivated purely by intellectual curiosity, and doesn’t really have too many practical application.  But it is pretty fun, and makes some neat looking pictures.  There is also some very beautiful math behind it, which I’ll now explain:

From 2D to 3D

It is a fundamental problem in computer vision to turn a collection of 2D images into a 3D model.  There are a number of ways to formulate this problem, but for this article I am going to focus on the following specialization of the problem which is known as*multiview stereo reconstruction*:

Given a collection of images of an unknown static object under known viewing configurations and lighting conditions, reconstruct the 3D object.

There are a few things about this which may seem a bit contrived; for example we might not always know where the camera is and what the lighting/material properties are like; however this are a still a large number of physical systems capable of taking these sort of controlled measurements.  One famous device is the [Stanford spherical gantry](http://graphics.stanford.edu/projects/gantry/), which can take photos of an object from points on a spherical shell surrounding.  In fact, this machine has been used to produce a number of [famous datasets](http://vision.middlebury.edu/mview/data/) like the well known “Dino” and “Temple” models:

[](http://vision.middlebury.edu/mview/data/)

One of the images from the temple data set acquired by the Stanford spherical gantry.

Of course even with such nice data, the multiview stereo problem is still ill-posed.  It is completely possible that more than one possible object might be a valid reconstruction.  When faced with such an ill posed problem, the obvious thing to do would be to cook up some form statistical model and apply Bayesian reasoning to infer something about the missing data.  This process is typically formulated as some kind of expectation maximization problem, and is usually solved using some form of non-linear optimization process.  While Bayesian methods are clearly the right thing to do from a theoretical perspective; and they have also been demonstrated to be [extremely effective at solving the 3D reconstruction problem in practice](http://vision.middlebury.edu/mview/eval/), they are also incredibly nasty.  Realisitic Bayesian models, ([like the ones used in this software, for example](http://grail.cs.washington.edu/software/pmvs/)), have enormous complexity, and typically require many hand picked tuning parameters to get acceptable performance.

Photo Hulls

All of this machine learning is quite well and good, but it is way too much overkill for our simple problem of reconstructing volumetric graphics from 8-bit sprites.  Instead, to make our lives easier we’ll use a much simpler method called *space carving:*

K.N. Kutulakos and S.M. Seitz, “[A theory of shape by space carving](http://www.cs.cmu.edu/~seitz/papers/kutu-ijcv00.pdf)” International Journal of Computer Vision (2000)

**UPDATE:**After Chuck Dyer’s comment, I now realize that this is probably not the best reference to cite.  In fact, a better choice is probably this one, which is closer to what we are doing here and has priority by at least a year:

S.M. Seitz, C. Dyer, “[Photorealistic Scene Reconstruction by Voxel Coloring](http://www.cs.cmu.edu/afs/cs.cmu.edu/user/seitz/www/papers/ijcv99.pdf)” International Journal of Computer Vision (1999)

(In fact, I should have known this since I took the computer vision course at UW where Prof Dyer works.  Boy is my face red!) **/UPDATE**

While space carving is quite far from being the most accurate multiview stereo algorithm, it is still noteworthy due to its astonishing simplicity and the fact that it is such a radical departure from conventional stereo matching algorithms.  The underlying idea behind space carving is to reformulate the multiview stereo matching problem as a fairly simple geometry problem:

**Problem:**Given a collection of images at known configurations, find the largest possible object in a given compact region which is visually consistent with each view.

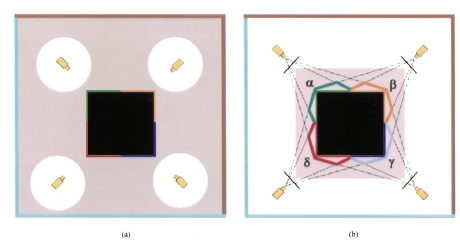
If you see a crazy definition like this, the first thing you’d want to know is if this is even well-defined.  It turns out that the answer is “yes” for suitable definitions of “visually consistent”.  That is, we require that our consistency check satisfy few axioms:

1. *Ray Casting*:  The consistency of a point on the surface depends only on the value of the image of each view in which it is not occluded.
2. *Monotonicity:* If a point is consistent with a set of views S, then it is also consistent with any subset of the views S' \subseteq S.
3. *Non-Triviality:*If a point is not visibile in any view, then it is automatically consistent.

It turns out that this is true for almost any reasonable definition of visual consistency.  In particular, if we suppose that our object has some predetermined material properties (ie is Lambertian and non-transparent) and that all our lighting is fixed, then we can check consistency by just rendering our object from each view and comparing the images pixel-by-pixel.  In fact, we can even say something more:

**Photo-Hull Theorem (Kutulakos & Seitz):**The solution to the above problem is unique and is the union of all objects which are visually consistent with the views.

Kutulakos and Seitz call this object the *photo hull* of the images, by an analogy to the [convex hull](http://en.wikipedia.org/wiki/Convex_hull) of a point set.  The photo hull is always a super set of the actual object, as illustrated in Fig. 3 of the above paper:

[](https://0fps.files.wordpress.com/2012/09/screenshot.png)

The photo hull illustrated. On the left is the physical configuration of an object and several cameras placed around it. On the right is the reconstructed photohull of the object. Note that both the left and right object produce identical projections. (c) 2000 Kutulakos&Seitz

If you stare at the above picture enough, and try looking at some of the examples in that paper you should get a pretty good idea of what the photo hull does.  Once you have some intuition for what is going here, it is incredibly simple to prove the above theorem:

**Proof:** It suffices to show that the photo hull, V^*, is visually consistent.  So suppose by contradiction that it is not; then there exists some point p \in V^*.  But by definition, there must be some visually consistent set V \subset V^* containing p \in V.  However, since V is a strict subset, the number views in which p is visible is a super set of the views containing it in V^*, and so by monotonicity and locality V can not be visually consistent.  Therefore, V^* is photo consistent and its uniqueness follows from the boundedness of the containing region.

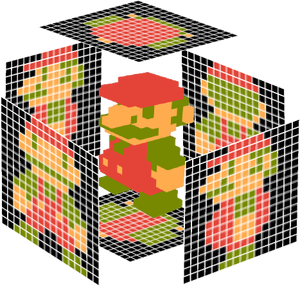
Carving Out Sprites

So now that we know what a photo hull is and that it exists, let’s try to figure out how to compute it.  It turns out that the above proof is highly suggestive of an algorithm.  What you could do is start with some large super set containing the object, then iteratively carve away inconsistent bits of the region until all you are left with is the maximal visually consistent set; or in other words the photo hull.  One simple way to accomplish this is to suppose that our region is really a voxel grid and then do this carving by a series of plane sweeps, which is essentially what Kutulakos and Seitz describe in their paper.

To make all of this concrete, let’s apply it to our problem of carving out voxel sprites.  The first thing we need to do is figure out what material/lighting model to use.  Because we are focusing on retro/8-bit sprites, let us make the following assumption:

**Flat Lighting Model:**Assume that the color of each voxel is contant in each view.

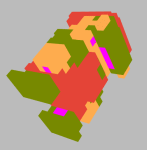
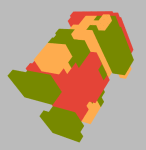
While this is clearly unphysical, it actually applies pretty well to sprite artwork, especially those drawn using a limited pallet.  The next thing we need to do is pick a collection of viewing configurations to look at the sprite.  To keep things simple, let us suppose that we view each sprite directly along the 6 cardinal directions (ie top, bottom, front, back, left, right) and that our cameras are all orthographic projections:

[](https://0fps.files.wordpress.com/2012/09/orthoview.png)

Putting this all together with the above sweep line method gives us a pretty simple way to edit sprites directly

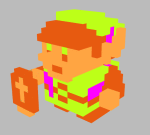
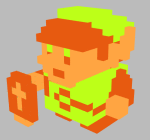
Filling in the Gaps

Of course there is still one small problem.  The shape we get at the end may have some small gaps that were not visible in any of the 6 views and so they could not be reconstructed.  To get a decent looking reconstruction, we have to do something about these missing points.  One simple strategy is to start from the edges of all these holes and then work inwards, filling in the missing pixels according to the most frequently occuring colors around their neighbors.  This produces pretty good results for most simple sprites, for example:

[](https://0fps.files.wordpress.com/2012/09/mario_no_fill.png)    [](https://0fps.files.wordpress.com/2012/09/mario_fill.png)

Left: No in painting.  Right: With in painting

Unfortunately, the limitations become a bit more apparent near regions when the image has some texture or pattern:

[](https://0fps.files.wordpress.com/2012/09/link_no_flle.png)      [](https://0fps.files.wordpress.com/2012/09/link_fill.png)

Left: No in painting.  Right: With in painting.  Note that the patterns on the left side are not correctly matched:

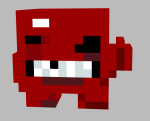
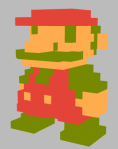
A better model would probably be to using something like a Markov random field to model the distribution of colors, but this simple mode selection gives acceptable results for now.

Editor Demo

As usual, I put together a little WebGL demo based on this technique.  Because there isn’t much to compare in this experiment, I decided to try something a bit different and made a nifty interactive voxel editor based on these techniques.  You can try it out here:

[http://voxelsprite.0fps.net](http://voxelsprite.0fps.net/)

There are a few neat features in this version.  In particular, you can share your models with others by uploading them to the server, for example:

[](http://voxelsprite.0fps.net/?src=/uploads/50564b03243216d714000007.png)     [](http://voxelsprite.0fps.net/?src=/uploads/50562d83243216d714000002.png)       [](http://voxelsprite.0fps.net/?src=/uploads/50562d60243216d714000001.png)

Left-to-right:  [Meat boy](http://voxelsprite.0fps.net/?src=/uploads/50564b03243216d714000007.png), [Mario](http://voxelsprite.0fps.net/?src=/uploads/50562d83243216d714000002.png), [Link](http://voxelsprite.0fps.net/?src=/uploads/50562d60243216d714000001.png)

How to use the editor

Here is a break down of the user interface for the editor:

[](https://0fps.files.wordpress.com/2012/09/sprite_editor.png)

And here is what all the buttons do:

1. *3D View:*  This is a 3D view of the photo hull of your object.  To move the viewing area around, left click rotates, middle click zooms and right click pans.  Alternatively, if you are on a mac, you can also use “A”, “S”, “D” to rotate/zoom/pan.
2. *Fill Hidden:*If you want to see what it looks like without the “gap filling” you can try unchecking the “Fill Hidden” feature.
3. *Sprite Editors:* On the upper left side of the screen are the editor panes.  These let you edit the sprites pixel-by-pixel.  Notice that when you move your cursor around on the editor panes, it draws a little red guide box showing you what subset of the model will be affected by changing the given pixel.
4. *Show Guides:* If the guide boxes annoy you, you can turn them off by deselecting “Show Guides” option.
5. *Paint Color:* This changes the color for drawing, you can click on the color picker at the bottom of the editor.  There is also an “eye dropper” feature to select another color on the sprite, where if you middle/shift click on a pixel in the sprite editor, then it will set the paint color to whatever you have highlighted.  Note that black (or “#000000”) means “Do not fill”.
6. *Palette:*To simplify editing sprite art, there is also a palette feature on the page.  You can set the paint color to a palette color by clicking on your palette; or you can save your paint color to the palette by shift/middle clicking on one of the palette items.
7. *Reconstruct Selected Views:*You can use this feature to automatically reconstruct some of the views of the object.  To select the views, click the check boxes (7a) and then to fill them in click “Reconstruct Selected Views” (7b).  This can be helpful when first creating a new object.
8. *Import Image:*Of course if editing images in a web browser is not your cup of tea, you can easily export your models and work offline (using tools like MSPaint or Photoshop) and then upload them to the browser when you are done.  The “Import Image” button lets you save the shape you are currently working on so you can process it offline.
9. *Export Image:*The “Export Image” button is the dual of the dual of “Import Image” and lets you take images you built locally and upload them to the server for later processing.
10. *Share:*This button lets you share your model online.  When you click it, a link will show up once your model has been uploaded to the server, which you can then send to others.
11. *Print in 3D:*This button uploads your model to ShapeWays so you can buy a 3D miniature version of your model in real life (more below).

There are also some more advanced features:

* *Undo:*  To undo the last action, press Ctrl+Z
* *Eye Dropper*:  To select a color on the sprite, you can either shift/alt/ctrl/middle/right click a color on the sprite editor.
* *Save to Palette:*  To save your current paint color to the palette, you can shift/alt/ctrl/middle/right click a palette entry.
* *Screenshot:*  To take a screen shot of your model, press “p”
* *Fullscreen:*“f” toggles fullscreen mode
* *Pan/Zoom:*Middle click or holding “s” zooms in and out.  Right click or holding “d” pans the camera.
* *Inconsistent Pixels:*Are highlighted in magenta on the sprite editor pane.
* *Sweep Preview:* Selecting views in the sprite editor filters them from the 3d object.

The source code for the client is available on [GitHub](https://github.com/mikolalysenko/mikolalysenko.github.com/tree/master/ShapeCarving), so feel free to download it and try it out.  (Though be warned!  The code is pretty ugly!)  Also please post any cool stuff you make in the comments!

3D Printing

The last cool thing that I added was the ability to 3D print your models on the ShapeWays store once you are done.  This means that you can take your designs and turn them into nifty 3D miniatures:

[](https://0fps.files.wordpress.com/2012/09/mario_minature.jpg)

A 3D printed miniature.

The resulting miniatures are about 5cm or 2in tall, and are super cool.  You can also look at the gallery of all the stuff everyone has uploaded so far on ShapeWays at the official store:

<http://www.shapeways.com/shops/0fps>

For every object that gets printed using this service, $10 gets donated to help support this blog.  If this fee is too expensive for you, the client is open source and you can basically copy the models out of the browser and print them yourself.  However, if you like the content on this site and want to help out, then please consider making a $10 donation by ordering a printed miniature through the store interface.  Not only will you be helping out, but you’ll also get a cool toy as well!

Acknowledgements

Thanks to everyone on facebook and #reddit-gamedev for early feedback.  Also, thanks to [ShapeWays](http://www.shapeways.com/) for providing the 3D printing service.  The client uses [three.js](https://github.com/mrdoob/three.js/) for rendering and [jsColor](http://jscolor.com/) for color picking.  The backend is written in [node.js](http://nodejs.org/) using [mongodb](http://www.mongodb.org/), and relies on [ShapeWays.JS](https://github.com/mikolalysenko/ShapeWaysJS) to interface with ShapeWays’ service.  Mario and Link are both copyrighted by Nintendo.  MeatBoy is copyrighted by  Team Meat.  Please don’t sue me!