

Tool Postmortem: Climax Brighton's Supertools

Pick a random game development studio, and take a guess as to what software you will find their artists using. Adobe Photoshop, for sure. Probably also 3DS Max or Maya, with a small minority running Lightwave or Softimage. At the extremes, you might even run into the odd copy of Houdini or Deluxe Paint.

Those guesses would be badly wrong if you happened to visit Climax Brighton, though. Here, we use a trio of inhouse tools called SuperModel, SuperTed, and the Bastard Love Child.

Why?

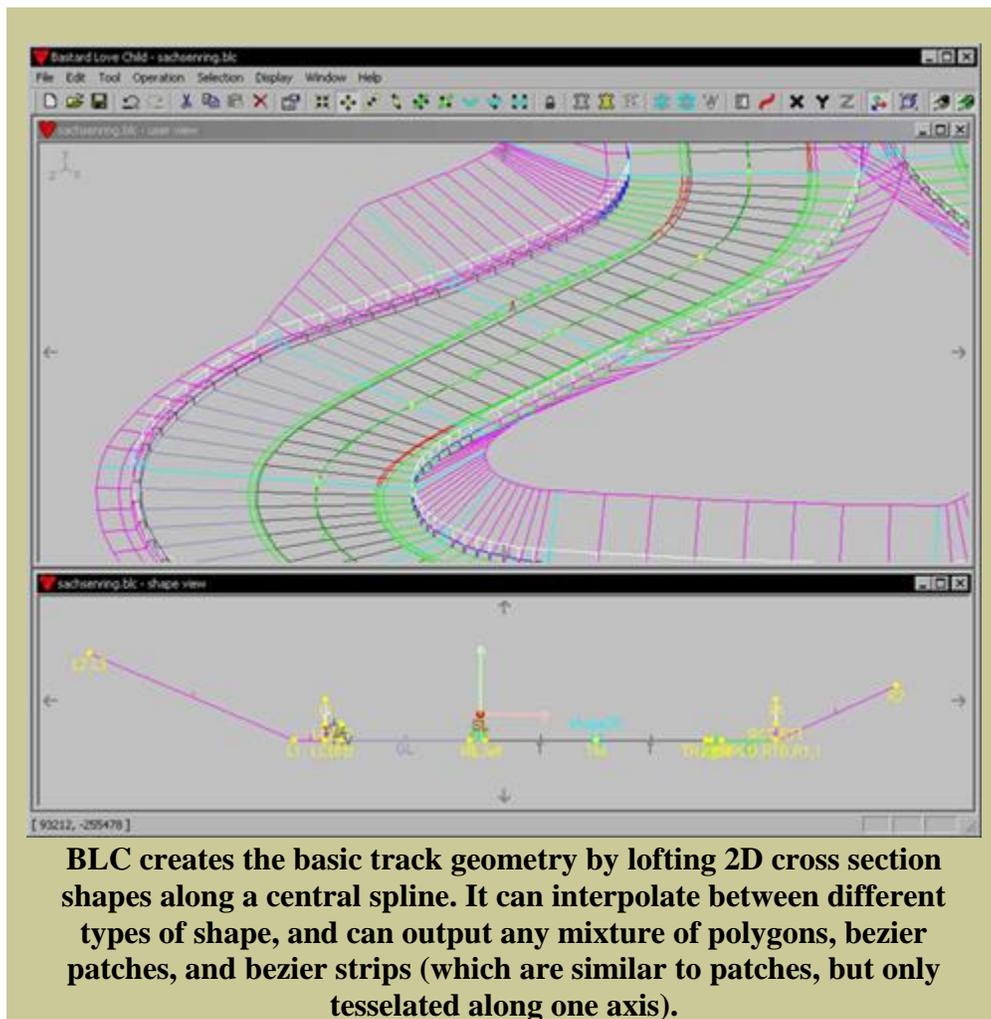
Mostly laziness, I think. We had a lot of graphics to build in not much time (familiar story?) and we didn't fancy the idea of working weekends, so we needed tools that would get the job done with a minimum of fuss. Hence Bezier Loft Creator (BLC, affectionately known as Bastard Love Child), a simple, specialised lofting package that quickly churns out the basic geometry for race tracks, and SuperTed, which lets us apply textures to those tracks using the smallest possible number of mouse clicks.

We started out making environments in BLC, buildings and vehicles in Maya, and texturing everything in SuperTed. But all was not well...



MotoGP: Ultimate Racing Technology (Xbox and PC) was one of the first games to be built using these art tools.

In an ideal world, everyone involved in making a game would be equally good at design, art, and coding, but few such multitalented individuals actually exist. Most programmers can get along fine without being able to create artwork (to the extent that 'programmer art' has become a common description for primary colored placeholder graphics), but artists are not usually so lucky. They may not actually be required to write C++, but they often run into sentences like "although we refer to a dependency graph as a singular graph, be aware you can display two or more independent graphs of connected nodes in the same window" (from the "Maya Essentials" manual). We have a couple of artists who are fine with this sort of thing (one even knows some C++), but we also have plenty who are not, and rightly so. We hired them because they had a good understanding of color and shape and how to make things look good, not because they knew how to use a computer!



The artists, being lazy, did not want to spend the next few hundred years learning how to use Maya properly. The programmers, also being lazy, would have loved to ignore this problem, but were unfortunately unable to do so due to our belief that the artists should be free to concentrate on the artwork.

We spent several months writing plugins that tried to make Maya behave a bit more sensibly, but the artists were still unhappy, and writing plugins is hard work. Also, our engine was doing some freaky things with curved surfaces (originally bezier patches, but these quickly evolved into various more efficient homebrew variants), and we were having a horrible time writing a converter that could reliably turn bezier patch source artwork into the optimal format for our engine.

Life would be simpler if the artists built their models directly using the same type of curved surface that we could render most efficiently, but adding new geometry types to Maya turned out to be hard verging on impossible due to API bugs. So, to our considerable surprise, the fastest and easiest solution turned out to be writing our own modelling package from the ground up!

It took two coders, six weeks, much questioning of the artists, and a few sessions of watching them work to see which functions were used most often, combined with a healthy dose of plagiarism as we borrowed some of the best ideas from Max and Maya. Today, we still use Maya for animation and Photoshop for textures, but everything else is done with our own tools.

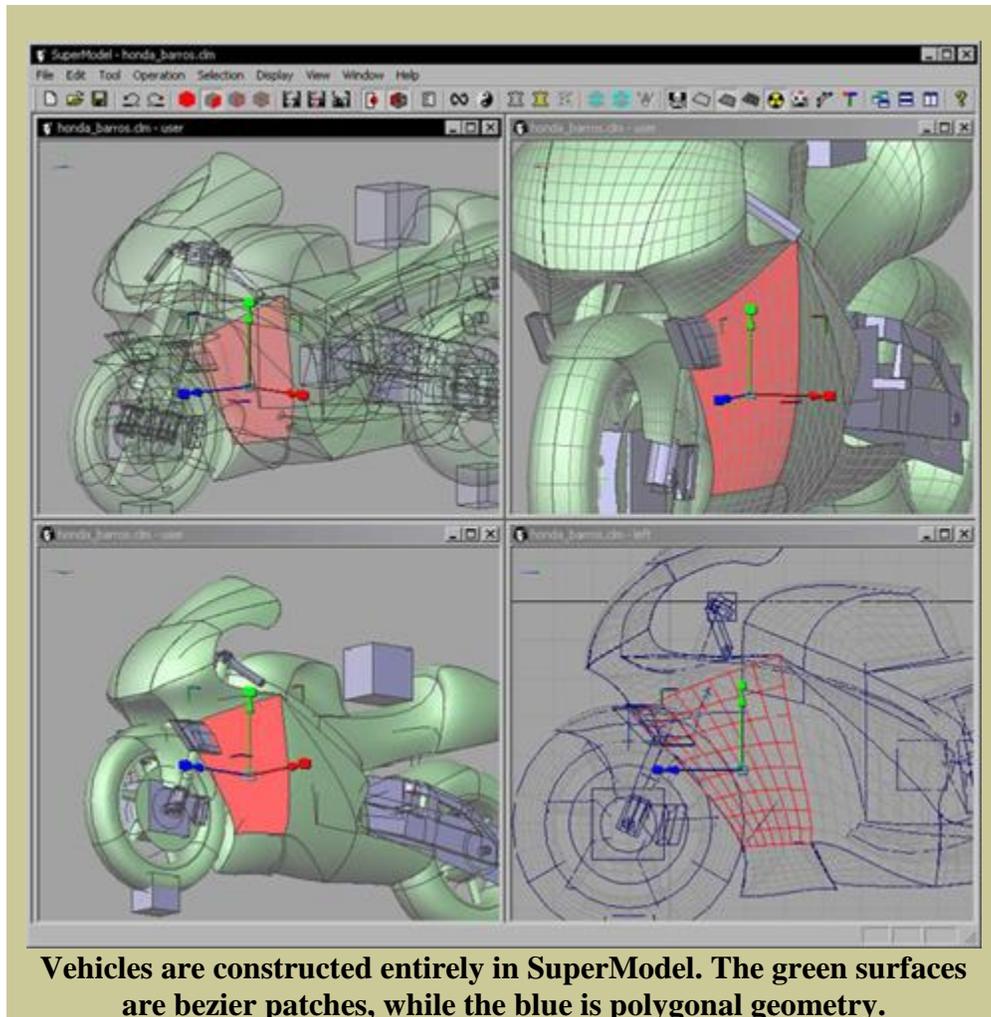
Approximate landscapes are quickly churned out using BLC, which is the bulldozer of the trio. It can shift huge amounts of geometry, but is crude and imprecise. SuperModel is more of a skilled craftsman, adding the buildings, instances, and other fine details, as well as correcting whatever things BLC got wrong. Finally, SuperTed handles texturing, lighting, and object/camera placement, bringing the final artwork up to a high state of polish.

Curved Surfaces

As hardware becomes more powerful, curved surfaces become ever more attractive. There just isn't enough time to position every single triangle by hand any more! But what sort of curved surface should you use, and how do you model with them?

A bezier patch is a rectangular area defined by sixteen control points. One at each corner, two along each edge, and four in the middle. They are simple, easy to model with, and can be efficiently implemented in hardware, but they are plagued by the problem of surface continuity. In simple terms this means that when you put two patches next to each other to build up a larger surface, they must be carefully arranged to make sure the surface will be smooth across the join.

The traditional solution is to use NURBS (Non Uniform Rational B-Splines). NURBS are a superset of the bezier patch, and allow large surfaces with many control points to be described as a single object which will always be perfectly smooth. After it has been constructed, a NURBS surface can be subdivided into multiple bezier patches for rendering, or at the simplest level, a NURBS object with the knot values (0, 0, 1, 1) is identical to a bezier patch.



The trouble with NURBS is that you simply cannot build efficient game artwork with them. Because they consist of a single large grid of control points, there is no way to add detail only where needed. For instance if you want an extra little curve in the nose of a character, you have to add a whole row of new control points running the entire way around the back of the head. Far too wasteful!

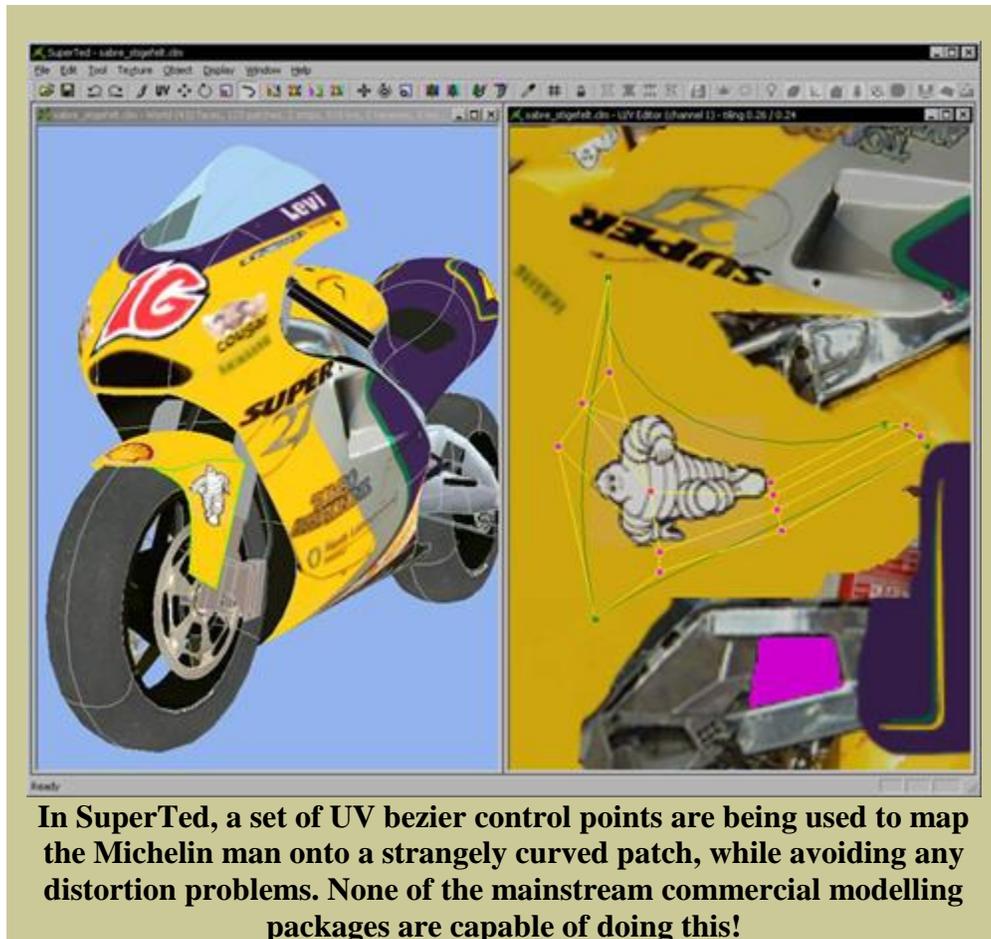
The big secret is that for games, surface continuity actually doesn't matter very much. As long as there are no holes between patches (which is trivial to arrange), and as long as the normals point in more or less the same direction along the join, it makes no difference if they are slightly wrong. Minor visual errors can be corrected by averaging the normals from adjacent patches, and even though this is mathematically bogus, who cares as long as nobody notices? The only thing that really matters is whether you can build good looking and efficient models quickly enough to get the game finished on time.

Efficient modelling requires that you use many small patches only in areas where detail is needed, and fewer large patches elsewhere. To connect the two areas, irregular triangular constructions will inevitably be required, but NURBS prevent that for the sake of maintaining an unnecessary level of surface smoothness. Using Maya, we had to work around the system by

creating every single patch as an individual NURBS surface, and writing our own plugins to line them up next to each other. This was awkward, and became pathologically slow for large models as Maya was never designed to have so many NURBS objects in one scene. Max does support arbitrary combinations of rectangular and triangular patches, but, again in the interest of maintaining good continuity, it hides access to the internal control points, which hugely limits the range of shapes that can be created.

When nothing else can do the job, the only option is to write your own. SuperModel would be useless for many tasks as it completely ignores the entire continuity issue, but for making games it is exactly what we needed.

Bezier curves aren't the only option, of course. Max and Maya both also support subdivision surfaces, which are appealing because they gracefully handle arbitrary types of mesh construction. Subdivs are less suitable for realtime tessellation, however, as they need too much information about the surrounding mesh, and they produce high output triangle counts from their recursive passes over the geometry, going up in steps of 4, 16, 64, 256, while patches can tessellate to 4, 9, 16, 25, and so on. The coarse nature of subdiv stepping is no problem for offline tessellation, where you can follow the subdivision with a progressive mesh style collapse back down to any desired polygon count, but one of the constraints on our choice was that the technique be appropriate for realtime implementation in the PS2 vector unit, for which bezier patches made a great deal more sense.



In most art packages, polygonal and curved surface modelling are entirely different modes, using different tools and having to be learned independently. In contrast, SuperModel lets the artists mix and match geometry types within a single object. If it curves along both axis, use a bezier patch. If it only curves in one axis, use a bezier strip. If it doesn't curve at all, use a polygon. It doesn't matter: everything is still made up of vertices and edges, which are manipulated using the same set of tools. If you extrude the edge of a polygon, another polygon will be created, but if you extrude the edge of a patch it will create another patch. Strips behave as either polygons or patches depending on which edge you select, so you can make half an object out of patches and the other half from polygons, while still having it entirely connected and all the edges welded together.

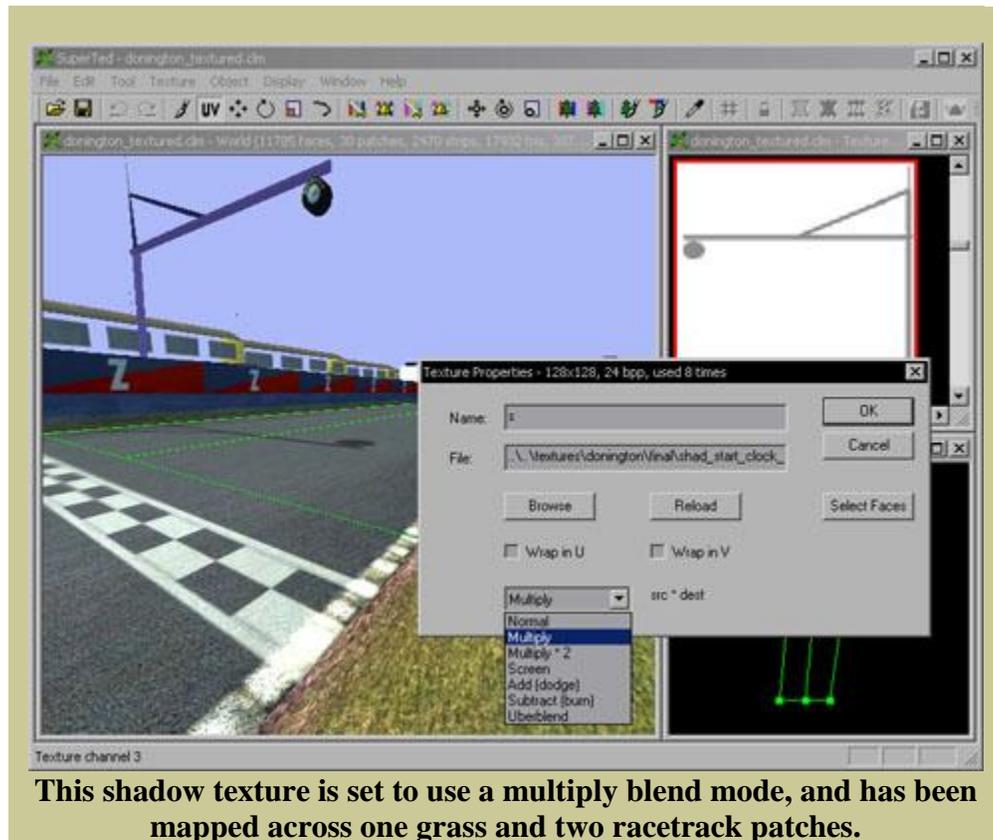
The more heavily a bezier model is optimised, the more difficult it becomes to texture, as the texture map will distort in strange ways when it is applied to the curved surface. Max simply ignores this problem (you have to live with the texture warping, or just not build such strangely curved patches in the first place), while Maya works around it by not supporting arbitrary UV mapping on curved surfaces. Not ideal for game use!

SuperTed solves this problem by supporting bezier interpolation for UV coordinates as well as for vertex positions. The UV control points can be automatically calculated to make a texture sit "flat" on a curved surface, or they can be edited by hand for particularly troublesome patches.

These tools allow us to accurately and efficiently texture geometry in ways that would be impossible with any other software.

Multitexturing

One texture per polygon just isn't enough any more. The PS2 has fillrate to spare, and the dual context rendering architecture can draw two copies of every triangle for little more cost than one, so you are wasting the hardware if you have only a single texture stretched over your geometry. Xbox supports four layers, and Gamecube eight (although in practice you can only afford to use two or three while maintaining a good framerate).



Modern gamers expect to see individual blades of grass and bits of gravel along with a decent amount of variety in the world, but although the hardware can easily draw this, consoles lack the memory to hold too many large textures. If you don't have room to store it, you'll have to synthesise it at runtime. Take a low resolution basic texture, add a layer of high frequency but repetitive detail, and over the top of that add some dirt or scratches. Reuse your set of dirt textures in different combinations over various base textures, and you can construct a massive number of variations from only a few building blocks.

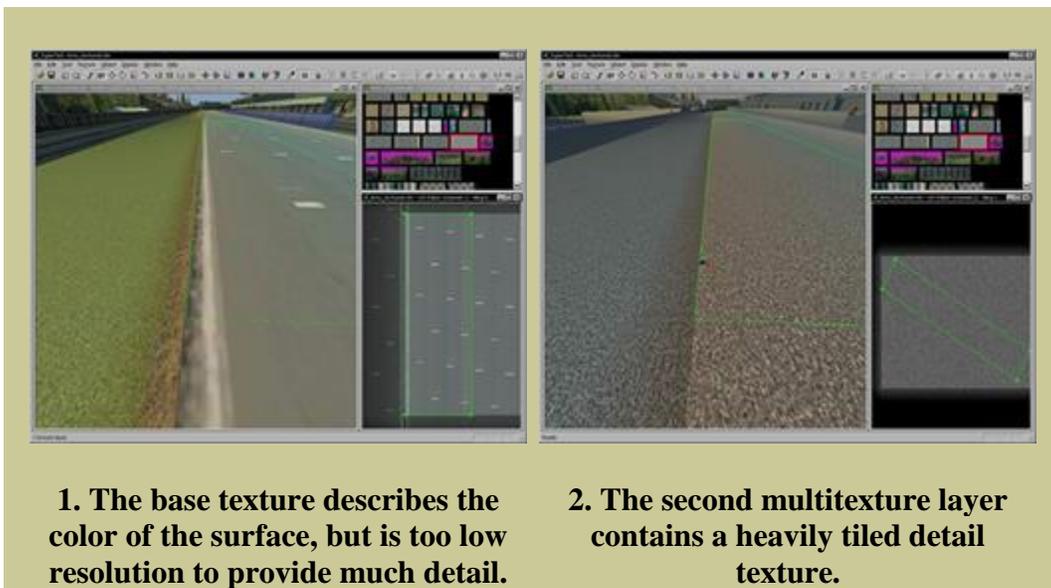
The nice thing is that this way of working turns out to be intuitive and easily understood by artists, as it is similar to the way they would have traditionally constructed a texture using multiple layers in Photoshop.

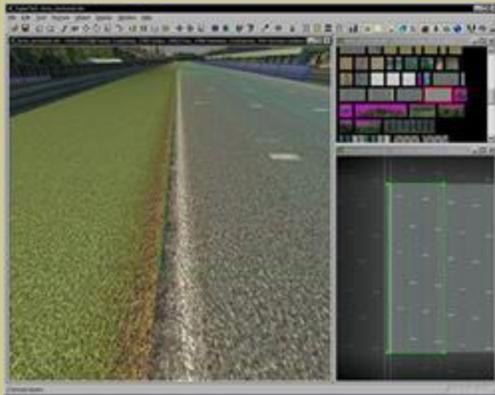
Most multitexture editors work by creating materials which combine a number of textures and blend modes, and then mapping these composite materials onto faces. In contrast, SuperTed works entirely with individual textures. It supports anywhere from one to sixteen layers, which can be textured independently, and every texture specifies how it is to be blended with the layer below it. This approach encourages the artists to reuse textures in creative ways, for instance adding in a bit of sand to make the side of a building look dusty, or UV mapping a single black circle to approximate the shadows from a huge variety of objects.

A few statistics might be in order here. Donington, one of the tracks in *MotoGP*, uses 352 textures, which are arranged into 602 unique combinations. Although the artists only choose one of seven possible blend modes for each texture, with three layers in use *MotoGP* ended up requiring a total of 84 different pixel shaders, 62 of which are used on the environment.

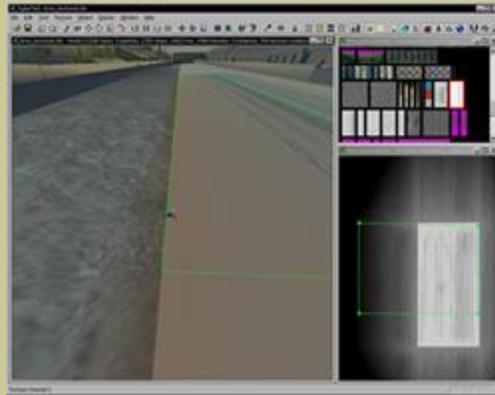
The downside to this variety of layers is that one patch is unlikely to be using the same set of textures as the ones around it, so our geometry tends to strip rather poorly. But I don't care: we may not be winning prizes for raw triangle throughput, but as long as we can manage a reasonable amount of geometry, I think it is more important to keep every surface looking as rich and interesting as possible.

This layer based approach to multitexturing does a good job of scaling across differences in hardware capabilities. Our PS2 projects are using two texture layers, with the gouraud alpha controlling a crossfade between them. On Gamecube we add a third multiply mode detail layer, while on Xbox the flexibility of pixel shaders lets the artists choose any possible combine modes for their three layers, with the fourth generally reserved for the programmers to do dynamic lighting or reflections.

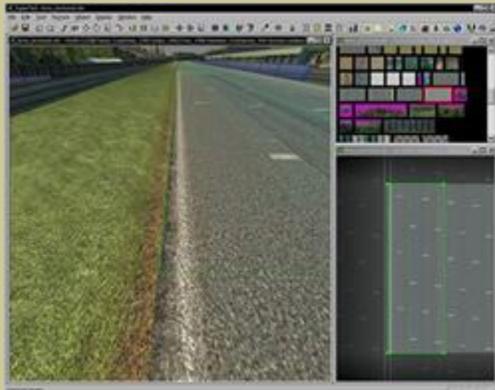




3. Multiplying the base and detail textures together gives an image many times more detailed than we would ever have room to store directly. The artists can combine layers in any way they like, so the same detail texture can be used over tarmac, road with painted lines, concrete areas, etc.



4. The third texture channel holds a coarse "scrotox" layer to make the grass look more lumpy, and skidmark overlays on the track. The artists have chosen to use a multiply mode for the scrotox layer, but regular alpha blending for the skid textures.



5. All three layers combined together in SuperTed.



6. Three artist-mapped texture layers, along with some per-pixel lighting on the fourth layer, running in-game on Xbox.

What Went Right

The tools are small, simple, shallow, and specialised. This is good for the artists because it keeps them fast, reliable, and easily learned, and good for the coders because it makes them practical to write and maintain. As several of us came from a Unix background, we liked the idea of many small tools each doing one task well and communicating through a shared file format, rather than having a single big program that tries to do everything all at once.

Our tools match the game engine in the big areas, but steer well clear of it in the little details. This allows the artists to work directly with the same types of curved surface and multitexturing technique that the game itself is using, while avoiding the need to update the editors for every minor game feature. Most attributes are mapped as text strings in the names of faces or objects, so the tools do not need explicit knowledge of what collision attribute we use to implement the 60 mph speed restriction in pitlanes, or the meaning of the pass condition on the wheelie box in one of the training missions.

Being in control of the entire art pipeline allowed for some surprising shortcuts. Collision attributes (grass, gravel, tarmac, or whatever) are stored in the name of each face. If these are marked on the initial BLC cross section shape at the start of track construction, we found ourselves automatically getting almost perfect sets of collision attributes out the end. This is because subsequent BLC shapes tend to be created by copying and then modifying the first one, and SuperModel geometry changes often involve copying or extruding faces, all of which will preserve the name attributes from surrounding geometry.

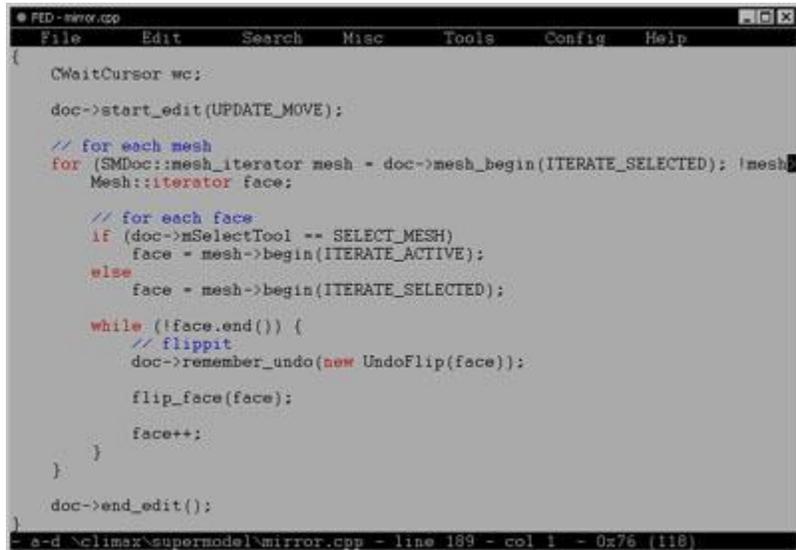
Likewise, the splines that were originally used to loft off the geometry in BLC came in handy later on for the AI and lap counting systems, with the cross sections that had been applied to them giving us precise information about the width of the track at each point. This reuse of data was only possible because the entire toolchain had been designed with our specific requirements in mind.

A final benefit of writing your own tools is the chance to give them stupid names, sounds, and icons. SuperModel plays cartoon spring noises when extruding a face, and explosions whenever you delete something. All three tools use a picture of a zebra skin as the icon for "hide selected", a tiger skin for "hide unselected", and a cow skin for "unhide". I suppose that's what you get for asking an artist to come up with three different variants of a "hide" icon using only the standard 16 color Windows palette! Silly, but fun.

What Went Wrong

These tools were developed and refined by one team over the course of a couple of games, but are now being used on many other projects across the company, which has led to inevitable disagreements about what features are needed. Everyone agrees that small, fast and specialised is the way to go, but no two teams have quite the same opinion as to exactly what we should specialise in! For instance *MotoGP* made little use of instances, and so the toolchain is not especially good at placing, texturing, and lighting instanced models, which has made life difficult for projects that are working more with instanced geometry.

Having multiple users raises the inevitable question of how the tools should be supported and maintained. The obvious solution is for each team to be in charge of adding whatever extra features they need, but they won't be familiar with the codebase or the overall design. There is a danger that one team could add features which get in the way or confuse things for others, and if too much is added we risk losing the advantages of small size, speed, and reliability. Perhaps we should set up a dedicated tools



```

C:\FED - mirror.cpp
File Edit Search Misc Tools Config Help
{
    CWaitCursor wc;
    doc->start_edit(UPDATE_MOVE);

    // for each mesh
    for (SMDoc::mesh_iterator mesh = doc->mesh_begin(ITERATE_SELECTED); mesh
    Mesh::iterator face;

    // for each face
    if (doc->mSelectTool == SELECT_MESH)
        face = mesh->begin(ITERATE_ACTIVE);
    else
        face = mesh->begin(ITERATE_SELECTED);

    while (!face.end()) {
        // flip it
        doc->remember_undo(new UndoFlip(face));

        flip_face(face);

        face++;
    }
}
doc->end_edit();
}
a-d \climax\supermodel\mirror.cpp - line 189 - col 1 - 0x76 (118)

```

department outside any specific project, but that risks becoming too insular and losing touch with real world requirements (as I have seen happen at other companies). The art toolchain is important enough that it should ideally be designed by the most experienced people available, which usually means the leads on current live projects, and the tools need to be maintained by someone directly in touch with the needs of each project as they evolve. Perhaps every team should just fork off their own specialised versions of whatever previous tools are closest to their needs, but that risks massive amounts of duplicate effort. Trying to reconcile these requirements will be an interesting challenge for the future.

SuperTed in particular has become increasingly complicated as it evolved, with too many obscure modes and specialised options. This is to some extent inevitable, and not a huge problem because we can just throw it away when it gets too clunky to maintain (the advantage of having multiple small tools is that we'll only have to rewrite one part of the chain, with BLC and SuperModel continuing unchanged), but a better design could perhaps have lasted more years before running into problems.

A minor but irritating difficulty on *MotoGP* was that while SuperTed always displays patches at a constant number of subdivisions, the game itself used an offline tessellator to do aggressive optimisations based on the curvature of each patch. In some cases this caused texture mapping errors which were laborious to fix as the problem could not be seen in the art tool due to the difference in algorithms. The obvious solution would be to make SuperTed use the same tessellator as the game itself, but how to do this without breaking generality across multiple projects that may be using different techniques?

One truly stupid mistake was to link gouraud colors and alpha in the user interface. This made sense to me as a programmer (after all, the whole RGBA value just goes into the vertex stream as one 32 bit integer), but in practice we used them to hold completely unrelated information. The RGB contains pre-baked static lighting, while the alpha holds either a multitexture blend factor, surface opacity, or shininess control. We frequently had problems with people accidentally

overwriting the alpha data while trying to change the RGB colors for different weather conditions, or vice versa. This interface could be improved if it simply provided arbitrary numbers of RGB and scalar control channels. The packed RGBA format is a hardware detail which artists ought not to be concerned with.

All our tools use OpenGL, and they do little in the way of clever rendering, just throwing everything at the driver and letting it take care of the details. On the whole this works fine, but SuperTed can become unstable when too many large textures are loaded. I'd like to see if a more careful texture management system could take some of the strain off the GL driver, and possibly even switch over to Direct3D. Although I'm a huge fan of GL for its ease of use in editor applications, a D3D implementation would make it easier to reuse code between the tools and a PC or Xbox game engine.

A recurring suggestion is to support multiplayer networked texturing, so that several people could work on the same file at the same time. This would be incredibly cool, but not useful enough to justify spending too much time on it :-)

We currently only support rectangular bezier patches, approximating triangles as quads with a zero length degenerate edge. This works fine as long as the tessellator fudges the normals to deal with the collapsed vertex, but modelling would be easier if we directly supported triangular patches. Their absence is a legacy from Maya (which only supports rectangles) and from doing realtime tessellation in the PS2 vector unit (which doesn't have enough instruction space to hold both triangle and rectangle microcode), but it would be good to support triangles for platforms that are doing offline tessellation.

For the future, I want to investigate more varied types of curved surface. N-Patches, modified butterfly subdivision surfaces, and displacement mapping all seem worth supporting, especially if they can be done in a way that would allow artists to mix and match multiple types of curved surface along with displacement maps within a single mesh.

The biggest improvement, though, will be some way to deal with arbitrary shader effects at both the vertex and pixel levels. Our current tools do an excellent job with any kind of static multitexturing technique, but dynamic effects such as animating meshes, reflections, and bumpmapping are only supported by flagging faces to which the game engine will later apply those effects. We could get better results if the artists were able to control things more directly, and especially if they could do this at a lower level than as entire shaders. In the same way that the current tools work with texture layers that can be used to build up any possible multitexture effect without needing any specific concept of "detail texture" or "lightmap", it would be great if the artists could build new surface types from shader fragments along the lines of "reflection cubemap", "projected texture", "animating texture", etc, and then be able to combine projected textures with animating textures to get moving cloud shadows. It would be easy enough to just pop up two edit boxes along with the caption "enter your vertex and pixel shader code here", but will be rather more of a challenge to present this kind of programmability in as artist friendly a way as the current Photoshop style texture layers system.

So, next time around I want to find a better solution to multiple projects having different requirements, to support more specialised and project specific types of surface tessellation, and to allow arbitrary shaders to be accessed directly inside the tools. This is all pointing towards a configurable, plugin based architecture, but surely that risks reintroducing all the layers of complexity that made us abandon Max and Maya in the first place? After all, they had good reasons for making their software so generic, and it would be unreasonable to assume I could build something equally flexible without running into the same problems of it becoming slow, unreliable, and hard to use. I do have a few ideas, though. Watch this space...