Pyro-Techniques: Playing With Fire

Remember that time your mom told you not to play with matches, but then you and your friends got together and accidentally burned down the neighbor's doghouse? In this month's column, we're going to build an even bigger fire, and we promise your mom won't ever find out.

Where There's Smoke...

First off, we need to identify exactly what kind of fire effect we want to generate. For our example, we'll be generating a bonfire in a real-time 3D environment that has a photorealistic art direction. Therefore, our fire effect will need to look as true-to-life as possible. Second, we'll sit down with the programmers and outline what tools and parameters we'll have to work with. In this example, we'll have access to and take advantage of the following: 32-bit texture maps up to 256×256 pixels, animatable sprites, RGB vertex color, vertex alpha, additive blending, and dynamic projection lighting. Last, we need to work with the programmers to generate an in-game particle system capable of pulling off the effect. In this case, we've prototyped the effect in an off-the-shelf product, 3D Studio Max, and the programming team has duplicated the required functionality demonstrated within Max's particle system.

Now we're ready to begin. There will be three main entities for our bonfire: flames, smoke, and sparks. Each will be created with groups of quad polygons, on which an animating sprite sequence has been mapped (in the case of the fire, this sequence will need to loop). The flames and smoke will be generated by clustering and overlapping these polygons in a random and chaotic manner. Why do we choose this method? Why not simply make a large polygon for the smoke and another for the fire? Because suspension of belief must be maintained as far as possible. Unless we have an inordinately long sequence of sprites, the viewer is easily going to be able to discern the looping pattern. Also, without some kind of random perturbation to the effect, every fire will look identical — boring. In Figure 1, you can see the effect we've outlined. On the left are the untextured polygons, with yellow representing the flames and blue representing the smoke. Notice that in the in-game shot on the right, it's difficult to identify the boundaries of a single individual polygon. This is possible because of how we've created our texture maps, which is the next step in the process.

Once the technique for creating the effect has been determined and the programmers have created the toolset, it is up to us as artists to build and iterate on the effect to get it looking just right. Before we start creating the textures, though, it's good to have some reference material from which to work. Fortunately, there are many graphical references for flames and explosions, as free downloads or from within pregenerated

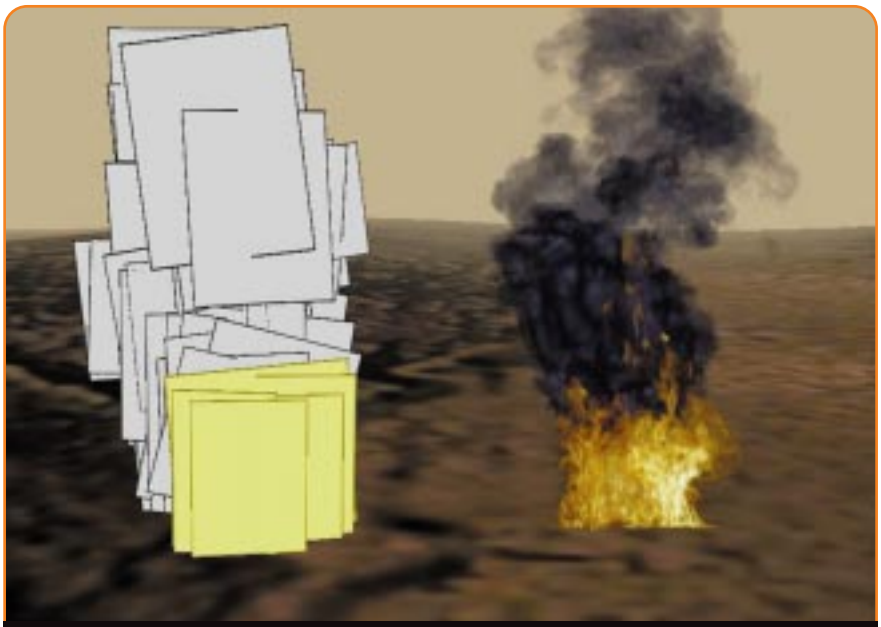


FIGURE 1. *Behind the scenes of the flame effect: the yellow polygons represent the flame, the blue ones represent the smoke.*

Mel has worked in the games industry for several years, and recently finished work as the art lead on DRAKAN. Currently, he manages a modeling and animation studio which provides custom content for RT3D games. Contact him at mel@infinexus.com.

FIGURE 2. *The reference graphic chosen for our bonfire.*

FIGURE 3. *A combination of a procedural texture and an alpha map make an effective smoke particle system (left). The in-game effect is shown at right.*

sets (Pyromania, ReelFire, and others). For our example, we'll be using texture reference from an off-the-shelf product, Pyromania 2, while the inspiration for the effect will come from a picture that was downloaded for free (Figure 2).

For the smoke entity of our effect, we'll use a nonlooping sequence of animated sprites 30 frames in length. (The length is arbitrary and should be tweaked to optimize the visual effect within the texture-memory requirements for the engine.) The smoke's color will be matched as closely as possible to that of the reference graphic, a dark bluish color. We will approximate the voluminous, billowing appearance of the smoke by the patterns in the texture, which were generated using a prerendered noise material in 3D Studio Max. Smoke has a fractal appearance, which means its edges appear soft and undefined — as you zoom into look at the edge, you keep finding more and more detail. To approximate this, all of the alpha maps used for the smoke will have soft, fuzzy edges. (In our particular case, the smoke's alpha components have been derived from a Pyromania 2 smoke effect.)

In Figure 3, you can see how the combination of using a procedural texture with a canned alpha map hides the individual texture boundaries, making the smoke particle system effective. On the right is the smoke effect as it will appear in the game. On the left you can see the results of using a texture with its originally-associated alpha map. Though the rendered effect for each individual texture is perfect, the textures have easily discernable boundaries, and when composited together resemble nothing if not a cluster of cotton balls. The smoke polygons are generated from random locations within the volume of the smoke emitter (this can be an arbitrary point in space, or can be defined by an artist as a geometric object). The speed at which the smoke animation is played, and each smoke polygon's vertical and horizontal speed, scale, and rotation, are all slightly randomized to achieve a less uniform look. When the textures on each polygon reach the end of their animated sequence, the polygon vanishes.

For the flames entity of our effect, we'll use a looping sequence of sprites 12 frames in length (as with the smoke, the sequence length is arbitrary). We'll match the flames' color with our reference graphic and the shape of the flames will be derived from a sequence of fire animations from Pyromania 2. It's important to note here that in the case of the smoke textures, we had several options available for generating the smoke textures and alpha maps. Smoke has a generally random, fractal pattern, and there are many programs that can generate realistic-looking cloud or smoke-like effects. In the worst case, I could have painted the smoke textures by hand in Photoshop or some other paint program with little or no degradation of the effect. For the fire, however, we're better off using an off-the-shelf, canned effect, at least as a starting point. The random, fractal nature of fire is extremely hard to duplicate by hand, and although current procedural methods for gener-

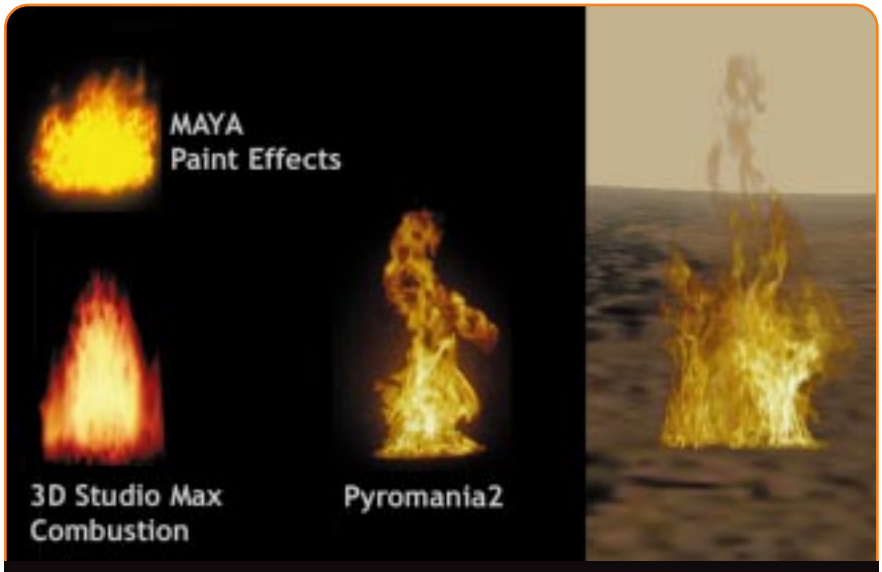
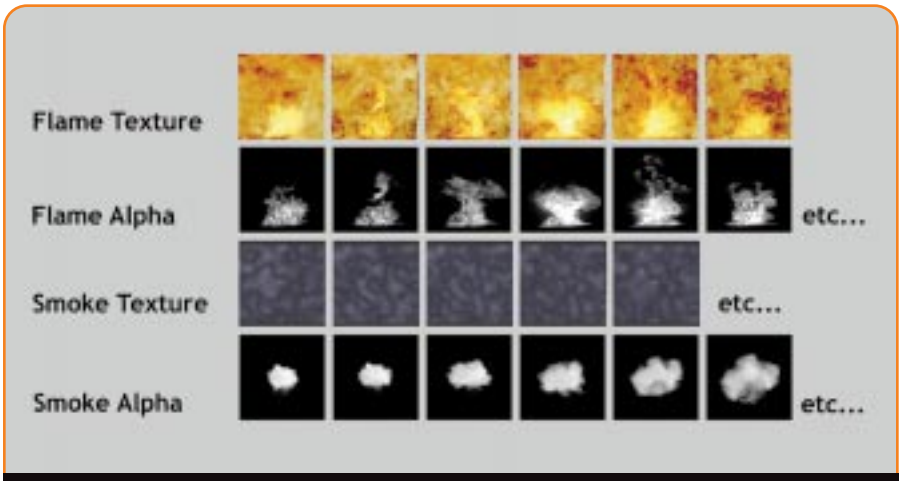


FIGURE 4. *A variety of fire effects generated by different software packages.*



Animated sprite sequences for our fire effect.

ating flames do a fair job, the effect falls far short of the real thing, as you can see in Figure 4. The first two groups of flames were generated by the standard flames particle systems in Paint Effects (Maya 2.5) and Combustion (3D Studio Max 3). As in the case of the smoke effect, the flame polygons will be generated with a random scaling factor, so that their horizontal and vertical size will vary slightly, although the lower edge of each polygon will be aligned. The flame polygons will also be different from the smoke polygons in that they will remain rooted to the ground throughout the entire effect. The rising nature of the flames, in this case, has been represented within the texture and alpha maps.

Adding the sparks entity to the effect will provide an added degree of randomness and break up the scale of the polygons involved (the polygons for the flames and smoke are about the same size to within a single order of magnitude). There is no need to generate additional textures for the sparks; textures from the flames and the alpha maps from the smoke can be combined for an excellent effect, and the colors will automatically match. The spark polygons are generated in much the same way as those for the smoke, though the scale is much smaller and the velocity much higher. With the textures created and in place, we've got a pretty decent bonfire going. Now it's time to use the remaining non-texture-based tools in our toolbox: vertex colors, vertex alpha, additive blending, and dynamic lighting.

Vertex Colors and Vertex Alpha

The vertices on each polygon can be colored to add graphic variation without additional texture expense. Most mainstream production tools support vertex-color editing. In our case, however, the vertex colors will work just as well. In any case, the vertex colors for our fire will be adjusted procedurally. I've found that this technique works best with the smoke polygons. Smoke is actually composed of fine particles that receive light and cast shadows. Consequently, if the smoke is thick enough, its color will actually be affected by light cast by the fire — the smoke will be lit from beneath as it rises. To achieve this, the vertices at the bottom of each smoke polygon will be varied as a function of their height from the fire; the closer to the fire, the more drastic the effect. In our case, the lowest vertices of each smoke polygon will be colored reddish-orange in accordance with the color of light being cast by the fire. In Figure 5 you see the effect. Note that to make full use of this effect, the smoke polygons have been subdivided. This is a level-of-detail option that can be turned off as the player gets farther away from the effect. Depending on the engine and implementation, you may need to lighten the smoke textures to see the effect of vertex colors. Although we've applied this effect only to the smoke polygons, the same procedure can be used to enhance or alter the colors in the flame and sparks polygons as well.

With the random generation of particles, there is always the problem that a smoke polygon will be created which goes off on its own, too far from its siblings. If this happens, the viewer may easily discern the polygonal nature of the effect and the suspension of disbelief will be lost. To minimize the impact of this, the errant polygon can be forced to fade out early. This can be done on a per-object level, but a more seamless effect can be achieved by sequentially fading out the individual vertices of the polygon as it crosses a defined threshold.

Additive Blending

Fires cast light, which is why we can see them even from a great distance. And though we can't effectively replicate this within the confines of a pixel-based 3D environment, we can certainly fake it. In order to make the flames appear to cast light or glow, the flame pixels must be arbitrarily brighter than the pixels surrounding them. For us,

FIGURE 5. *Adding vertex colors to the smoke polygons (shown at right) gives the smoke the appearance of being lit by the fire beneath it.*

this is equivalent to saying the flame pixels must be arbitrarily more white. In fact, the brightest we can ever make an object appear is to force all its pixels to read RGB = 255,255,255.

Additive blending can achieve this for us. When pixels are assigned to have additive blending, they increase the brightness, or whiteness in this case, of the pixels onto which they are blended. Thus, by compositing these pixels in several layers, each succeeding layer creates a brighter, whiter effect.

This is a perfect application for our flame polygons. Since they are clustered and overlapping, the densest portion of the flames will have the brightest (most white) pixels. This is exactly what we want to achieve, since it will help us mimic the light-casting nature of real-life flames. Figure 6 shows the result of additive blending in contrast with the previous version of the flame effect. (This effect can even be further improved by adding a slight halo or lens flare to the flames.)

Dynamic Projection Lighting

The light cast from a fire jumps and dances in concert with the chaotic motion of the flames. In order to approximate this effectively in the game, we need to use a projection light. The actual implementation of this effect can vary widely with each engine, but in essence the projection light should derive its pattern from the same sequence of alpha maps as that of the flame polygons.

When this is done correctly, the flames will "dance" on any surface within the range of the light. The effect of the dancing lights, although difficult to represent in a static image, can be stunning in game, and will greatly enhance the realism of the effect. The final in-game product of our fire effect, incorporating all of the above techniques, is shown in Figure 7.

FIGURE 6. *Additive blending techniques have been added to the flame pixels, creating the effect seen at right.*

FIGURE 7. *Our in-game scene, showing our final fire effect.*

Fanning the Flames

Many of the effects we used in this month's example were not possible a few years ago. The technology simply wasn't there, or didn't have enough of a user base to be applicable on the target platform. However, the pace of technology shows no sign of slowing down and we are always being presented with more tools and methods for content generation. We need to ask ourselves constantly "What if?" so we'll be better prepared to apply new technologies and become more creative and efficient developers. ■