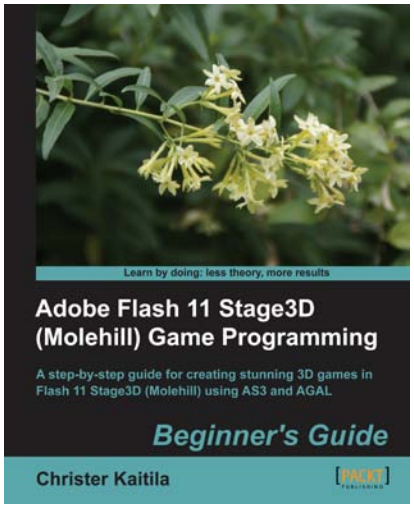
[PACKT] PUBLISHING

Adobe Flash 11 Stage3D (Molehill) Game Programming Beginner's Guide

Christer Kaitila



Chapter No. 8

"Eye-Candy Aplenty!"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.8 "Eye-Candy Aplenty!"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Christer Kaitila, B.Sc., is a veteran video game developer with 17 years of professional experience. A hardcore gamer, dad, dungeon master, artist, and musician, he never takes himself too seriously and loves what he does for a living: making games! A child of the arcade scene, he programmed his first video game in the Eighties, long before the Internet or hard drives existed.

The first programming language he ever learned was 6809 assembly language, followed by BASIC, Turbo Pascal, VB, C++, Lingo, PHP, Javascript, and finally ActionScript. He grew up as an elite BBS sysop in the MS-DOS era and was an active member of the demo scene in his teens. He put himself through university by providing freelance soft ware programming services for clients. Since then, he has been an active member of the indie game development community and is known by his fellow indies as Breakdance McFunkypants.

Christer frequently joins game jams to keep his skills on the cutting edge of technology, is always happy to help people out with their projects by providing enthusiastic encouragement, and plays an active part helping to find bugs in Adobe products that have not yet been made public. Over the years, he has programmed puzzle games, multi player RPGs, action titles, shooters, racing games, chat rooms, persistent online worlds, browser games, and many business applications for clients ranging from 3D displays for industrial devices to simulations made for engineers.

He is the curator of a popular news website called `www.videogamecoder.com` which syndicates news from hundreds of other game developer blogs. He would love to hear from you on twitter `(www.twitter.com/McFunkypants)` or Google+ `(http://www.mcfunkypants.com/+)` and is always happy to connect with his fellow game developers.

His client work portfolio is available at `www.orangeview.net` and his personal game development blog is `www.mcfunkypants.com` where you can read more about the indie game community and his recent projects.

He lives in Victoria, Canada with his beloved wife and the cutest baby son you have ever seen.

This book would not have been possible without valuable contributions of source code, tutorials and blog posts by Thibault Imbert, Ryan Speets, Alejandro Santander, Mikko Haapoja, Evan Miller, Terry Paton, and many other fellow game developers from the ever-supportive Flash community. Thank you for sharing. Your hard work is humbly and respectfully appreciated.

Adobe Flash 11 Stage3D (Molehill) Game Programming Beginner's Guide

Adobe's Stage3D (previously codenamed Molehill) is a set of 3D APIs that has brought 3D to the Flash platform. Being a completely new technology, there were almost no resources to get you acquainted with this revolutionary platform, until now.

This book will show you how to make your very own next-gen 3D games in Flash. If you have ever dreamed of writing your own console-style 3D game in Flash, get ready to be blown away by the hardware accelerated power of Stage3D. This book will lead you step-by-step through the process of programming a 3D game in ActionScript 3 using this exciting new technology. Filled with examples, pictures, and source code, this is a practical and fun-to-read guide that will benefit both 3D programming beginners and expert game developers alike.

Starting with simple tasks such as setting up Flash to render a simple 3D shape, each chapter presents a deeper and more complete video game as an example project. From a simple tech demo, your game will grow to become a finished product—your very own playable 3D game filled with animation, special effects, sounds, and tons of action. The goal of this book is to teach you how to program a complete game in Stage3D that has a beginning, middle, and game over.

As you progress further into your epic quest, you will learn all sorts of useful tricks such as ways to create eye-catching special effects using textures, special blend modes for transparent particle systems, fantastic vertex and fragment programs that are used to design beautiful shaders, and much more. You will learn how to upload the geometry of your 3D models to video RAM for ultra-fast rendering. You will dive into the magical art of AGAL shader programming. You will learn optimization tricks to achieve blazingly fast frame rate even at full screen resolutions. With each chapter, you will "level up" your game programming skills, earning the title of Molehill Master—you will be able to honestly call yourself a 3D game programmer.

This book is written for beginners by a veteran game developer. It will become your trusty companion filled with the knowledge you need to make your very own 3D games in Flash.

What This Book Covers

Chapter 1, Let's Make a Game Using Molehill! In this chapter, we talk about what Stage3D (Molehill) is, what it can do, and the basic terminology you need to know when dealing with 3D games.

Chapter 2, Blueprint of a Molehill. In this chapter, we compare the differences between old-fashioned Flash games and the new way of doing things, along with a description of the major classes we will be working with and the structure of a typical Stage3D game.

Chapter 3, Fire up the Engines. In this chapter, we take the first step by setting up our tools and programming the initializations for our game. The result is a demo that gets Stage3D to animate a simple 3D mesh.

Chapter 4, Basic Shaders: I can see Something! In this chapter, we learn about programming shaders using AGAL and adding text to the display. The result is an upgraded demo with four different animated shaders.

Chapter 5, Building a 3D World. In this chapter, we create a way to fill the game world with complex 3D models by programming a mesh data file parser. The result is a game demo complete with high-poly spaceships and terrain instead of simple textured squares.

Chapter 6, Textures: Making Things Look Pretty. In this chapter, we upgrade our game demo to include a keyboard input and special render modes that allow us to draw special effects such as transparent meshes, explosions, and more. The result is a demo that highlights these many new effects.

Chapter 7, Timers, Inputs, and Entities: Gameplay Goodness! In this chapter, we program a timer and generic game entity class. In addition, we upgrade the GUI with a heads-up-display overlay and add a chase camera. The result is a demo with a spaceship that can fl y around in an asteroid field that looks more like a real video game.

Chapter 8, Eye-Candy Aplenty! In this chapter, we program a highly optimized GPU particle system for use in special effects. All geometry is rendered in large, reusable batches. The result is a game demo that is able to render hundreds of thousands of particles at 60fps.

Chapter 9, A World Filled with Action. In this chapter, we upgrade our game engine to include simple game actor artificial intelligence, collision detection, and a map parsing mechanism to allow for easy world creation. The result is a fully functional engine ready for use in a real game.

Chapter 10, 3... 2... 1... ACTION! In this chapter, we add the final touches to our game project such as a title screen, dealing with the score, damage and game over events, music and sound, and much more. The final result of our efforts is a fully playable 3D shooter game filled with action!

Appendix A, AGAL Operand Reference. This appendix provides operand references that have been used in this book.

Appendix B, Pop Quiz Answers. In this section, we provide answers to the pop quiz.

8
Eye-Candy Aplenty!

You have reached Level 8.

The next step in our quest for graphical glory is to implement some eye-candy. Particle systems and special effects are essential for giving your game a visual impact.

Things are getting really exciting now. You can probably see the light at the end of the tunnel already: our game is taking shape and already we have learned so much. Now that we have achieved some basic gameplay—movement, animation, a GUI heads-up-display, timers and input classes—the next step is to make it look good.

In this chapter, we are going to implement a system capable of pumping out massive amounts of special effects. There are serious performance considerations to take into account—after all, we don't want our special effects to destroy the silky smooth framerate.

Our current quest

With optimization and abstract reuseability in mind, let's create a particle system manager capable of rendering insane numbers of particles while having little effect upon the framerate. We want to be able to use it to pump out everything from explosions, smoke, sparks, shock waves, blood splatters, lens flares, splashes of water or lava, and atmospheric motes of dust floating around in the air.

What we want is to be able to render TONS of particles with great framerate:

Before we dive in, we should take note of some design goals for our fancy new particle system.

Designing for performance

Many Flash 3D game engine authors lately have been having trouble achieving good particle system performance. The reason is that our first instinct is to simply draw thousands of billboard sprites individually. Of course, it turns out that there is overhead in calculating the transform matrix and render state for each draw call and the only way we can achieve large numbers of particles in Flash is to "batch" them in larger chunks. Even better, if we can avoid doing any of the animation simulation on the CPU and offload all calculations on the GPU using AGAL, our particle engines can benefit from virtually no CPU use.

We want a particle system that allows mega complex particle systems of nearly infinite numbers of polies. One possible solution is to create a specialized vertex buffer for the entire particle system.

With this technique, all particle system animation would be handled on the GPU with virtually no rendering overhead for animation of gigantic particle systems. Why? The entire thing is rendered in ONE draw call. Essentially, you are "pre-calculating" the entire animation for your explosions and other effects. No worrying about each and every sprite in an explosion, you just draw the whole group at the same time and get the GPU to do all the heavy lifting.

By using this technique, we will be able to implement a fully GPU animated particle system (60fps with 40,000 particles each moving in different directions, rotating, and scaling larger while fading out) that uses AGAL for all simulation—not the CPU.

Designing for reusability

We don't want to hard-code our special effects rendering for any particular kind of effect. Therefore, we will aim for the ability to send any kind of mesh to the system: from single billboard sprites consisting of a basic quad to an entire group of polygons that make up a complex explosion.

Another form of reuseability is the ability to reuse previously created particles over and over. We also don't want to bog down the CPU and eat up tons of RAM with each frame by creating and destroying thousands of meshes over and over. Doing so will kill the framerate because the time it takes to upload mesh data (or even create simple AS3 objects) adds up—if we are going to be spawning particles over and over during the game there is no reason to continually initialize them and then destroy them when they disappear.

Therefore, apart from doing all simulation on the GPU, another optimization technique used here is to create a "particle pool", so that particles are reused if inactive. This avoids any GC (garbage collection) issues which result in framerate hiccups: we don't "spawn" new particles each time there is a new explosion, we just reuse old ones and only create new ones if there are not any inactive ones available. The result of doing so is that the framerate stays near 60fps, whether we have forty thousand particles on the screen or none at all.

Animating using AGAL

The AGAL and scene setup for the technique outlined earlier is relatively simplistic. We create two vertex buffers. One contains the start positions and the other the end positions, pre-calculated, for each vertex. For example, instead of creating thousands of GameEntities, create a single 10,000 poly "mesh" with vertex buffer data as follows: x1,y1,z1,x2,y2,z2. Instead of moving each and every particle around through AS3 matrix math, the motion of the sprite is pre-calculated and stored in the vertex buffer itself.

Once we have created this particle "batch geometry", we can send a delta value as a vertex buffer constant register that varies from 0..1 over time. In each frame, when we render the mesh, we will have your AGAL vertex program interpolate the x,y,z position of each vertex from the starting to the ending position. Each vertex will smoothly move from the first to the second position.

By pre-calculating the motions of each particle once and then simply getting AGAL to smoothly move from one location to the next, we can simulate gravity, wind, the expanding size of things such as smoke dissipating into the air, and more.

In order to achieve great performance during gameplay, we don't want to do any complex calculations in AS3. Instead, once we have filled the two vertex buffers described earlier, all we need to do for each frame is instruct Stage3D how much time has passed. Before we render our particle mesh, we send this value to Stage3D for use in the shader.

The key design consideration here is to NOT render or simulate each particle separately. Depending on your computer's horsepower (and in particular your video card's fill rate), it is entirely feasible that your game can boast scenes containing 100,000+ particles while still running at 60fps—with room to spare for all your other scene models, as we are not using the CPU for any of the heavy lifting.

In order to successfully complete this quest, we need to perform the following:

- Create a basic particle entity class
- Program the AGAL for keyframed vertex animation
- Code a particle system manager to control multiple particles
- Design some nice looking effect art
- Incorporate the particle system into our game

Time to upgrade our game's eye-candy. This efficient "batched GPU particle system" technique will give you the pizazz required for some awesome screenshots and in-game action sequences. Let's get started!

A basic particle entity class

The first step in adding eye-candy to our game is to define what a particle is. Eventually, our game will spawn hundreds of particles for use in all sorts of special effects, so let's make it abstract and able to be used for numerous kinds of particles—from splashes of water to sparks that fly from clashing swords.

One particle in our special effects system can be a single polygon or a premade "batch" of many polygons. As we want to have many polies with minimal framerate overhead, we are going to treat entire effects (such as an explosion) as a single entity even though the mesh it uses will consist of hundreds or thousands of polies.

We want our particle entity class to hold mesh data, as well as a texture and shader. This sounds suspiciously similar to the `Stage3dEntity` class we have already programmed. The great news is that we only need to add a few minor extras to this basic entity class, so instead of creating a brand new kind of AS3 object from scratch, we will simply extend the basic entity class and add a few extra functions and variables.

Time for action – extending the entity class for particles

Create a new file in your source code folder named `Stage3dParticle.as`. Import the basic functions we will need and simply extend the base class as follows:

```
// Game particle class version 1.0
//
package
{

import com.adobe.utils.*;
import flash.display.Stage3D;
import flash.display3D.Context3D;
import flash.display3D.Context3DProgramType;
import flash.display3D.Context3DTriangleFace;
import flash.display3D.Context3DVertexBufferFormat;
import flash.display3D.IndexBuffer3D;
import flash.display3D.Program3D;
import flash.display3D.VertexBuffer3D;
import flash.display3D.*;
import flash.display3D.textures.*;
import flash.geom.Matrix;
import flash.geom.Matrix3D;
import flash.geom.Vector3D;

public class Stage3dParticle extends Stage3dEntity
{
```

What just happened?

The last line in the preceding code instructs Flash to "extend" or take all the functions that were implemented in our `Stage3dEntity` class and assume they are the same in this new class. In this way, our particle class instantly has all the helper functions in its repertoire that we worked so hard on before.

This use of class inheritance helps save a lot of time and effort by allowing us to reuse the code. It keeps basic functionality in one place. If we upgrade this base class or fix a bug, then all classes that extend it automatically get upgraded as well.

Time for action – adding particle properties

Next, we want to give our new class a few extra properties related to particles. We do this as follows:

```
public var active:Boolean = true;
public var age:uint = 0;
public var ageMax:uint = 1000;
public var stepCounter:uint = 0;

private var mesh2:Stage3dObjParser;
private var ageScale:Vector.<Number> =
  new Vector.<Number>([1, 0, 1, 1]);
private var rgbaScale:Vector.<Number> =
  new Vector.<Number>([1, 1, 1, 1]);
private var startSize:Number = 0;
private var endSize:Number = 1;
// we only want to compile the shaders once
private static var particleshader1mesh:Program3D = null;
private static var particleshader2mesh:Program3D = null;
```

What just happened?

These properties will be used to simulate our particle systems. By keeping track of age, we can calculate the proper size and color over time, so that an explosion, for example, starts small and expands in size while fading out. The times are stored in milliseconds, so the defaults above will create a particle that exists for exactly one second.

Time for action – coding the particle class constructor

Now that we have defined the variables we need, let's write a new class constructor function. We don't want to use the one that comes with our original `Stage3dEntity` class, so we use the keyword "override" to replace it with this new version. Though it is similar to a basic entity, our particles can have an optional second mesh that is used for keyframed vertex position animation. We also don't need to pass in a shader as we will be defining a special AGAL shader later.

```
// Class Constructor - the second mesh defines the
// ending positions of each vertex in the first
public function Stage3dParticle(
  mydata:Class = null,
  mycontext:Context3D = null,
  mytexture:Texture = null,
```

```
    mydata2:Class = null
    )
  {
    transform = new Matrix3D();
    context = mycontext;
    texture = mytexture;

    // use a shader specifically designed for particles
    // this version is for two frames: interpolating from one
    // mesh to another over time
    if (context && mydata2) initParticleShader(true);
    // only one mesh defined: use the simpler shader
    else if (context) initParticleShader(false);

    if (mydata && context)
    {
      mesh = new Stage3dObjParser(
        mydata, context, 1, true, true);
      polycount = mesh.indexBufferCount;
      trace("Mesh has " + polycount + " polygons.");
    }

    // parse the second mesh
    if (mydata2 && context)
      mesh2 = new Stage3dObjParser(
        mydata2, context, 1, true, true);

    // default a render state suitable for particles
    blendSrc = Context3DBlendFactor.ONE;
    blendDst = Context3DBlendFactor.ONE;
    cullingMode = Context3DTriangleFace.NONE;
    depthTestMode = Context3DCompareMode.ALWAYS;
    depthTest = false;
  }
```

What just happened?

This constructor simply sets up a particle entity to be rendered with the correct blend mode. If we pass a second mesh in the parameters, that mesh is also parsed and ready for use in our shader. Depending on whether this is a particle that uses a single mesh or interpolates in shape from one to the next, the appropriate shader is created.

Time for action – cloning particles

As we only need to compile the AGAL shader or upload mesh geometry to Stage3D once for each kind of particle, we avoid wasting RAM by creating a special clone function:

```
public function cloneparticle():Stage3dParticle
{
  var myclone:Stage3dParticle = new Stage3dParticle();
      updateTransformFromValues();
  myclone.transform = this.transform.clone();
  myclone.mesh = this.mesh;
  myclone.texture = this.texture;
  myclone.shader = this.shader;
  myclone.vertexBuffer = this.vertexBuffer;
  myclone.indexBuffer = this.indexBuffer;
  myclone.context = this.context;
  myclone.updateValuesFromTransform();
  myclone.mesh2 = this.mesh2;
  myclone.startSize = this.startSize;
  myclone.endSize = this.endSize;
  myclone.polycount = this.polycount;
  return myclone;
}
```

What just happened?

All this function does is copy the variables needed to a new copy of the current particle. This is one of the various optimizations that will allow our game to render TONS of particles using minimal system resources.

Time for action – generating numbers used for animation

The next two functions are simply handy routines that you may want to use for particle simulation. They smoothly interpolate a value using the sine function such that it "wobbles" in and out. This function is really handy as you design particles that are supposed to bob up and down, or fade in and then fade out smoothly.

```
private var twoPi:Number = 2*Math.PI;
// returns a float from -amp to +amp in wobbles per second
private function wobble(
  ms:Number = 0, amp:Number = 1, spd:Number = 1):Number
{
  var val:Number;
  val = amp*Math.sin((ms/1000)*spd*twoPi);
```

```
    return val;
  }

  // returns a float that oscillates from 0..1..0 each second
  private function wobble010(ms:Number):Number
  {
    var retval:Number;
    retval = wobble(ms-250, 0.5, 1.0) + 0.5;
    return retval;
  }
```

What just happened?

What do these "wobble" functions do? They are used to generate numbers that smoothly change from zero to one (and possibly back again) based on how much time has passed. They are sent to the AGAL code in our particle shader to fade the particle in and out and to make it grow in size.

Time for action – simulating the particles

The next step is to code a `step` function. This function will eventually be called once every single frame, for every single particle, by your particle system manager class. Therefore, it is important for this function to be as simple as possible.

```
public function step(ms:uint):void
{
  stepCounter++;
  age += ms;
  if (age >= ageMax)
  {
    //trace("Particle died (" + age + "ms)");
    active = false;
    return;
  }
  // based on age, change the scale for starting pos (1..0)
  ageScale[0] = 1 - (age / ageMax);
  // based on age, change the scale for ending pos (0..1)
  ageScale[1] = age / ageMax;
  // based on age: go from 0..1..0
  ageScale[2] = wobble010(age);
  // ensure it is within the valid range
  if (ageScale[0] < 0) ageScale[0] = 0;
  if (ageScale[0] > 1) ageScale[0] = 1;
  if (ageScale[1] < 0) ageScale[1] = 0;
```

```
    if (ageScale[1] > 1) ageScale[1] = 1;
    if (ageScale[2] < 0) ageScale[2] = 0;
    if (ageScale[2] > 1) ageScale[2] = 1;
    // fade alpha in and out
    rgbaScale[0] = ageScale[0];
    rgbaScale[1] = ageScale[0];
    rgbaScale[2] = ageScale[0];
    rgbaScale[3] = ageScale[2];
  }
```

What just happened?

Using the time elapsed since the previous frame, this function uses the wobble functions to generate a list of numbers that smoothly go from zero to one, or one to zero, depending on what is required.

It first checks to see if the age of the particle has advanced beyond its desired lifetime. If so, the particle no longer needs to be updated or rendered. If it is instead still active, we calculate the `ageScale` and `rgbaScale` values which will be used by our shader to do the animation.

Time for action – respawning particles

As mentioned earlier, we only want to create brand new particles when there are not any inactive particles that could be reused instead. This is how we are able to get so many polies on the screen with such great performance. By avoiding creating and destroying new objects for every frame, we keep the memory requirements constant during the game loop.

```
  public function respawn(
    pos:Matrix3D, maxage:uint = 1000,
    scale1:Number = 0, scale2:Number = 50):void
  {
    age = 0;
    stepCounter = 0;
    ageMax = maxage;
    transform = pos.clone();
    updateValuesFromTransform();
    rotationDegreesX = 180; // point "down"
    // start at a random orientation each time
    rotationDegreesY = Math.random() * 360 - 180;
    updateTransformFromValues();
    ageScale[0] = 1;
    ageScale[1] = 0;
    ageScale[2] = 0;
```

```
    ageScale[3] = 1;
    rgbaScale[0] = 1;
    rgbaScale[1] = 1;
    rgbaScale[2] = 1;
    rgbaScale[3] = 1;
    startSize = scale1;
    endSize = scale2;
    active = true;
    //trace("Respawned particle at " + posString());
  }
```

What just happened?

The `respawn` function is used by our upcoming particle system manager to reset the variables required, so that an existing particle is ready for "a new life". Whenever our games need to create a new explosion, for example, we simply need to instruct an existing particle to start animating at a new location. We can also tweak the size or lifespan of our particle, if needed, so we can do such things as create bigger or smaller explosions depending on the amount of damage inflicted.

One important note: we rotate our explosion mesh to "point down" in this example because the example data in our second `.OBJ` file has each particle in a slightly higher location. Depending on your source art, you may not want to include this line. You might also want to enhance this routine to be able to turn off the random orientation, for example, which is used here to make each explosion look more different from the previous.

The `ageScale` and `rgbaScale` variables are initialized with values that make sense for the very first frame of our particle shader. We will be changing these values in every frame in order to fade in or grow the mesh by sending them to our AGAL shader for use as vertex and fragment constants.

Time for action – rendering particles

Just as we extended the default `Stage3dEntity` class constructor function, we also want to write our own particle rendering function. Override the `render` function as follows:

```
// optimization: reuse the same temporary matrix
private var _rendermatrix:Matrix3D = new Matrix3D();
override public function render(
  view:Matrix3D,
  projection:Matrix3D,
  statechanged:Boolean = true):void
{
  // only render if these are set
  if (!active) return;
```

```
if (!mesh) return;
if (!context) return;
if (!shader) return;
if (!texture) return;

// get bigger over time
scaleXYZ = startSize +
  ((endSize - startSize) * ageScale[1]);

//Reset our matrix
_rendermatrix.identity();
_rendermatrix.append(transform);
if (following) _rendermatrix.append(following.transform);
_rendermatrix.append(view);
_rendermatrix.append(projection);

// Set the vertex program register vc0
// to our model view projection matrix
context.setProgramConstantsFromMatrix(
  Context3DProgramType.VERTEX, 0, _rendermatrix, true);
```

We start by first updating the scale of our mesh to grow or shrink our particle depending on the values we passed to the respawn function. This simple scaling updates our transform matrix using the `ageScale` variable. It is then mixed with the view and projection matrix in the same way as we have been doing in all previous rendering examples.

```
// Set the vertex program register vc4
// to our time scale from (0..1)
// used to interpolate vertex position over time
context.setProgramConstantsFromVector(
  Context3DProgramType.VERTEX, 4, ageScale);

// Set the fragment program register fc0
// to our time scale from (0..1)
// used to interpolate transparency over time
context.setProgramConstantsFromVector(
  Context3DProgramType.FRAGMENT, 0, rgbaScale);
```

Next, we send our interpolation values to the shader. By filling a vertex program constant with `ageScale` and fragment program constant with `rgbaScale`, our Stage3D shader will be able to do the rest of the work for us. Remember that these two variables contain a smoothly interpolated list of numbers (four in each) that go from 1 to 0 or 0 to 1 over time as needed. The `step()` function is where these values are updated in each frame.

```
    // Set the AGAL program
    context.setProgram(shader);
    // Set the fragment program register ts0 to a texture
    context.setTextureAt(0,texture);
    // starting position (va0)
    context.setVertexBufferAt(0, mesh.positionsBuffer,
      0, Context3DVertexBufferFormat.FLOAT_3);
    // tex coords (va1)
    context.setVertexBufferAt(1, mesh.uvBuffer,
      0, Context3DVertexBufferFormat.FLOAT_2);
    // final position (va2)
    if (mesh2)
    {
      context.setVertexBufferAt(2, mesh2.positionsBuffer,
        0, Context3DVertexBufferFormat.FLOAT_3);
    }

    // set the render state
    context.setBlendFactors(blendSrc, blendDst);
    context.setDepthTest(depthTest,depthTestMode);
    context.setCulling(cullingMode);

    // render it
    context.drawTriangles(mesh.indexBuffer,
      0, mesh.indexBufferCount);
  } // render function ends
```

What just happened?

The particle render function sends numbers based on how much time has passed to our special particle shader, so that the mesh will be rendered at the correct size and transparency. These numbers are calculated in the `step()` function and are stored in AGAL registers for use in our vertex and fragment programs.

Finally, we instruct Flash which vertex buffers and shader to use, set the render state so that it uses transparency properly (in this example, additively brightening the scene), and render our mesh.

There is only one missing function in our particle class—the AGAL shader—which deserves a more detailed discussion.

Keyframed vertex animation shader

The final touch required to complete our particle class is to program the shader that does all the heavy lifting during our game loop. In the constructor function for our particle class, we call this `initParticleShader` function and it returns a shader that uses either two meshes (for keyframed animation), or just one if that is all that is needed.

As mentioned in previous chapters, a Stage3D shader is made up of a vertex program and a fragment program. The former handles the positions, UV coordinates, and more for each vertex in your mesh, while the latter decides what color and transparency each pixel on the screen is when your mesh is rendered.

As we want to get your video card's GPU to do all the matrix math for the movement of our particles (freeing up the CPU to think about other things), we need to write an AGAL vertex program that can move your mesh's vertexes around over time.

One way to do this is to store the final positions of each vertex alongside the start positions and simply interpolate from one value to the next over time. An easy way to set this up is to sculpt an explosion, for example, in whatever 3D modeling application you prefer. Then, after exporting the first version of your mesh as an `.OBJ` file for use in your game, scale, rotate, and move the shape around and export this second mesh.

This is a very simplistic form of "keyframe" animation. Frame one of your animations is the first mesh and frame two is the second. The shader morphs the shape of our mesh from the first to the second over time.

This two-mesh technique is optional: if you wish, you can pass only one mesh to your constructor and simply scale the entire mesh as a whole (using the parameters of the `respawn` function) while fading it out. However, for added motion (and so that each particle in your effect can move independently rather than as a whole) this two-frame animation approach is easy to achieve and results in great-looking visuals.

Time for action – creating a keyframed particle vertex program

Add the following to the very end of your `Stage3dParticle` class:

```
private function initParticleShader(twomodels:Boolean=false):void
{
  var vertexShader:AGALMiniAssembler =
    new AGALMiniAssembler();
  var fragmentShader:AGALMiniAssembler
    = new AGALMiniAssembler();
  if (twomodels)
```

```
        {
          if (particleshader2mesh)
          { // already compiled previously?
             shader = particleshader2mesh;
             return;
    }
          trace("Compiling the TWO FRAME particle shader...");
          vertexShader.assemble
          (
            Context3DProgramType.VERTEX,
            // scale the starting position
            "mul vt0, va0, vc4.xxxx\n" +
            // scale the ending position
            "mul vt1, va2, vc4.yyyy\n" +
            // interpolate the two positions
            "add vt2, vt0, vt1\n" +
            // 4x4 matrix multiply to get camera angle
            "m44 op, vt2, vc0\n" +
            // tell fragment shader about UV
            "mov v1, va1"
          );
        }
```

What just happened?

The preceding AGAL code is used to interpolate the vertex positions of mesh 1 with the vertex positions defined by mesh 2. In our render function, we use the function `setVertexBufferAt` to set up va0 to contain the first vertex buffer and `va2` as the second. We also sent the variable `ageScale`, the list of values that smoothly change from 0 to 1 over time to Stage3D, using the `setProgramConstantsFromVector`. This four number list now resides in the vc4 vertex constant register, ready for use.

For each vertex, the preceding shader multiplies the xyz position of the first mesh (stored in va0) by the first number stored in vc4 (which in our example starts at 1 and gradually goes to 0). Therefore, the first line of AGAL above scales the location and stores this value in the temporary register vt0.

The second line of AGAL above does the same for the "frame two" mesh, except it multiplies it by the second number in `ageScale` (the y component of `vc4`) and stores it in the `vt1` temporary register.

Finally, these two-scaled locations (`vt0` and `vt1`) are added together and stored in `vt2`, which is now the interpolated value. Depending on how much time has passed, it represents a point somewhere between the first and second vertex position. This new location is a matrix multiplied by the model view projection matrix in the same way that all previous vertex programs have, so that our camera angle is taken into account. Finally, we pass the UV texture coordinates for each vertex in varying register `v1` for use in our fragment shader.

Time for action – creating a static particle vertex program

Continuing with this function, add the following:

```
else
{
  if (particleshader1mesh)
  { // already compiled previously?
      shader = particleshader1mesh;
      return;
  }
  trace("Compiling the ONE FRAME particle shader...");
  vertexShader.assemble
  (
    Context3DProgramType.VERTEX,
    // get the vertex pos multiplied by camera angle
    "m44 op, va0, vc0\n" +
    // tell fragment shader about UV
    "mov v1, va1"
  );
}
```

What just happened?

In the preceding code, if we are using a single mesh particle (with no keyframe animation), then the AGAL for the vertex shader is extremely simplistic: it just instructs Stage3D to account for the camera angle and stores the proper vertex position in the op (output position) register.

Time for action – creating a particle fragment program

Now that we have programmed an AGAL vertex program that can smoothly interpolate from one location to the next over time, we can do something similar in the fragment program to generate the proper pixel color on the screen. Continuing with the `initParticleShader` function:

```
// textured using UV coordinates
fragmentShader.assemble
(
  Context3DProgramType.FRAGMENT,
  // grab the texture color from texture 0
  // and uv coordinates from varying register 1
  // and store the interpolated value in ft0
  "tex ft0, v1, fs0 <2d,linear,repeat,miplinear>\n" +
  // multiply by "fade" color register (fc0)
  "mul ft0, ft0, fc0\n" +
  // move this value to the output color
  "mov oc, ft0\n"
 );
```

What just happened?

In the AGAL above, we create a fragment program that samples the texture that our render function defined as `fs0` at the proper location based on the UV vertex coordinates that our vertex program passed in varying register `v1`. The color of the texture at this point is stored in a fragment program temporary register `ft0`.

Before rendering using this value, we want to be able to smoothly fade out over time, so that our explosions or smoke puffs don't pop out abruptly, but instead fade to nothingness. In our render function, we sent the variable `rgbaScale` to Stage3D in `fc0`, a fragment constant register. Just as the timescale used in our vertex program, the four numbers stored in `fc0` will range from zero to one to zero again—perfect for a nice smooth fade.

In the second line of AGAL above, we simply multiply the texture color that we got in the first step by the fading value stored in `fc0`. We then store the final, properly faded out color in the oc (output color) register.

Time for action – compiling the particle shader

Now that we have written the proper AGAL for use in our particles, we simply compile them together to form a shader and upload it to the graphics card.

```
        // combine shaders into a program and upload to the GPU
        shader = context.createProgram();
        shader.upload(
          vertexShader.agalcode,
          fragmentShader.agalcode);

        // remember this shader for reuse
        if (twomodels)
            particleshader2mesh = shader;
        else

        particleshader1mesh = shader;
      } // end initParticleShader function

    } // end class

    } // end package
```

That is it for our fancy new `Stage3dParticle` class! You could start using this class in place of our now-familiar Stage3dEntity in your game if you wanted to spawn a single unique particle here and there. As this class is derived from Stage3dEntity, you could even make it "follow" something else. This would be perfect if you wanted to attach a "glow" to a lamppost, for example.

This class is perfect for one-offs—particles that are meant to persist for a long time in your game. However, for temporary special effects such as puffs of smoke or blood splatters, we need a particle system manager capable of iterating through lists of hundreds of particles and reusing them over and over.

A particle system manager class

In addition to batching geometry and doing the vertex animation and fade-out in AGAL, the other great optimization used for achieving good particle performance has nothing to do with the Stage3D API specifically and is instead a standard game system optimization: a "pool" of particles that can be reused.

The second class we need to program in order to fill our game with the eye-candy we desire is a manager of numerous particles. This system will track time passing and will call the `step` and `render` functions for all active particles. Whenever our game needs to create a special effect of some kind, this particle system will get the job done.

For performance reasons, we don't want to be creating and destroying objects during the render loop, so instead we simply reuse inactive particles whenever possible. If the entire queue of particle entities is currently visible, only THEN do we create a new object.

By avoiding any variable allocations or destructions during the game, we never run into GC (garbage collection) issues. GC issues cause stutters in your game and reduce the framerate. They are caused by temporary variables that are no longer in use being destroyed by Flash. Even worse, if you don't minimize the creation of variables during game play, you will get what is called a "memory leak"—rising use of system RAM that never ends.

As the computer's memory becomes full of useless junk, the browser will eventually crash. This is why we have been careful to avoid the creation of too many temporary variables, and why we would never want to program a particle system that creates a new `Stage3dParticle` each time we want a special effect to take place—eventually there would be millions in memory, slowing things down.

By creating a class that intelligently reuses inactive objects that were created earlier, we save all these hassles and gain the benefit of much smoother framerates!

Time for action – coding a particle system manager class

This controller class will be much simpler than the particle class we programmed earlier. The particle system manager is simply a control mechanism that tries to reuse inactive particles whenever possible. Create a new file in your project folder named `GameParticlesystem.as` and define our new class as follows:

```
// Game particle system manager version 1.0
// creates a pool of particle entities on demand
// and reuses inactive ones whenever possible
//
package
{
import flash.utils.Dictionary;
import flash.geom.Matrix3D;
import flash.geom.Vector3D;
import Stage3dParticle;

public class GameParticlesystem
{
  // contains one source particle for each kind
  private var allKinds:Dictionary;
  // contains many cloned particles of various kinds
  private var allParticles:Dictionary;
  // temporary variables - used often
  private var particle:Stage3dParticle;
  private var particleList:Vector.<Stage3dParticle>;
  // used only for stats
  public var particlesCreated:uint = 0;
  public var particlesActive:uint = 0;
  public var totalpolycount:uint = 0;

  // class constructor
  public function GameParticlesystem()
  {
    trace("Particle system created.");
    allKinds = new Dictionary();
    allParticles = new Dictionary();
  }
```

What just happened?

In the preceding code, we simply define our new class and set up a few handy variables. The `allKinds` list contains the "master copy" of each new type of particle that we need for our game.

The `allParticles` list is where are the renderable particles are stored. This list will eventually be filled with hundreds of particles of different kinds, both active and inactive. For each frame we will check this list to see which particles need updating or rendering.

Time for action – defining a type of particle

```
// names a particular kind of particle
public function defineParticle(
  name:String, cloneSource:Stage3dParticle):void
{
  trace("New particle type defined: " + name);
  allKinds[name] = cloneSource;
}
```

What just happened?

The preceding function is meant to be called during your game initialization. We pass a name as well as a `Stage3dParticle` (which contains the mesh geometry, shader, texture, and the like). This particle will be used as the source for all future particles of that particular kind. None of these particles is ever rendered: they are simply associated with a string name such as "explosion" or "water splash" and are used to provide something to clone.

Time for action – simulating all particles at once

The `step` function is called in every frame during our game loop. In this way, our game needs only to instruct the entire system to step to ensure that each of the potentially thousands of particles is updated.

```
// updates the time step shader constants
public function step(ms:uint):void
{
  particlesActive = 0;
  for each (particleList in allParticles)
  {
    for each (particle in particleList)
    {
      if (particle.active)
      {
        particlesActive++;
        particle.step(ms);
      }
    }
  }
}
```

What just happened?

When running the particle system's `step` function, our game engine will need to pass the elapsed time in milliseconds following the previous frame. In this way, all particle animations are not framerate-dependent and will animate at the same rate regardless of system performance.

It loops through each particle type (based on name) in the `allParticles` list, and then each particle of that type that has been created so far. Any particle that is currently active is stepped forward in time. Each particle's `step` function we defined in our `Stage3dParticle` class earlier is run, so that we can update the movements and transparency as necessary.

Time for action – rendering all particles at once

As above, when the particle system needs to be rendered, we need to loop through every particle and instruct it to draw itself.

```
// renders all active particles
public function render(view:Matrix3D,projection:Matrix3D):void
{
  totalpolycount = 0;
  for each (particleList in allParticles)
  {
    for each (particle in particleList)
    {
      if (particle.active)
      {
        totalpolycount += particle.polycount;
        particle.render(view, projection);
      }
    }
  }
}
```

What just happened?

The particle system's `render` function iterates through all active particles and runs their corresponding `render` function, so that they are displayed on the screen. We are keeping track of how many polygons are rendered in each frame (primarily for boasting purposes).

Time for action – spawning particles on demand

The final function needed by our particle system manager class is responsible for either creating new particles or reusing old, inactive ones. Our game engine is going to use the `spawn` function to "ask" that a new particle be triggered.

```
// either reuse an inactive particle or create a new one
public function spawn(
  name:String, pos:Matrix3D, maxage:Number = 1000,
  scale1:Number = 1, scale2:Number = 50):void
{
  var reused:Boolean = false;
  if (allKinds[name])
  {
    if (allParticles[name])
    {
      for each (particle in allParticles[name])
      {
        if (!particle.active)
        {
          //trace("A " + name + " was reused.");
          particle.respawn(pos, maxage, scale1, scale2);
          particle.updateValuesFromTransform();
          reused = true;
          return;
        }
      }
    }
```

What just happened?

We first check to see if that particular kind of particle has been previously defined by name. If it is a known type, then we iterate through all previously created particles of that kind and look for an inactive one that is available for reuse. If we find one, we simply call its respawn function to reset some variables and get it animating again in the proper new location.

Time for action – creating new particles if needed

We need to account for situations where all available particles are currently active and none are available for reuse. Continuing with the particle system's `spawn()` function:

```
    else
    {
      trace("This is the first " + name + " particle.");
```

```
            allParticles[name] = new Vector.<Stage3dParticle>;
          }
          if (!reused) // no inactive ones were found
          {
            particlesCreated++;
            trace("Creating a new " + name);
            trace("Total particles: " + particlesCreated);
            var newParticle:Stage3dParticle =
              allKinds[name].cloneparticle();
            newParticle.respawn(pos, maxage, scale1, scale2);
            newParticle.updateValuesFromTransform();
            allParticles[name].push(newParticle);
          }
        }
        else
        {
          trace("ERROR: unknown particle type: " + name);
        }
      }

    } // end class

    } // end package
```

What just happened?

If no inactive particles of the desired type are available, then it is time to use up a little more memory and create a fresh new clone of our appropriate "master" particle. We make a new one and from this point forward, our `allParticles` list is a little bit bigger.

That is it for our fancy new particle system class. As you can see, by creating particles "on demand" we only create enough to satisfy the needs of our game engine. This technique has many advantages, including the fact that RAM consumption stays relatively constant once a certain number of particles have been created.

It is still entirely possible to bog down your computer by spawning so many particles that the system simply cannot handle the load. In particular, if you spawn too many particles in close succession, you might get into situations where all available particles are currently visible on the screen and new ones need to be created. This can happen when you give particles too long a `maxAge`, so that they exist in an active state for too long.

Even with this new optimized particle system solution, we have to be careful to only create as many particles as needed and not go overboard. If you are generating scenes with more than 100,000 polies used just for particles, then you should rethink your special effects. In particular, remember that even if your game is running smoothly on your power gaming PC, users trying to play it on wimpy netbooks (or tablets and smartphones) will not have the same amount of horsepower.

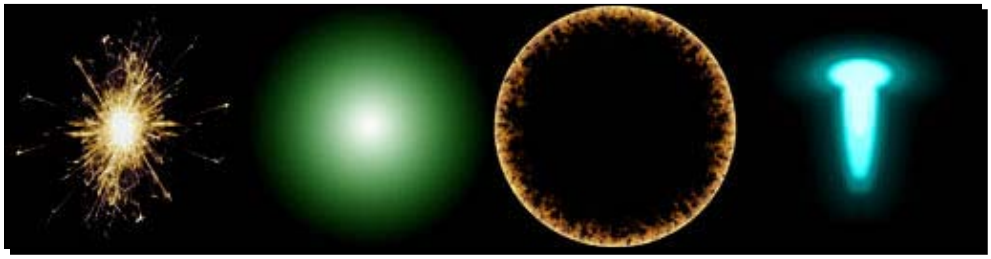
Perhaps you can get by with larger particles that use fewer polies, or make your particles fade out more quickly so fewer are on the screen at any one time. Finally, remember that one mesh with 500 polygons renders much faster than 500 meshes of one polygon each.

Keyframed particle meshes

In order to test our fancy new particle system, we need some cool-looking art assets to use. We will need a few new meshes and some transparent textures. The key technique here is the creation of two meshes for use as the keyframes for our particle system. We are aiming to create the "before" and "after" for each effect.

Selecting a particle texture

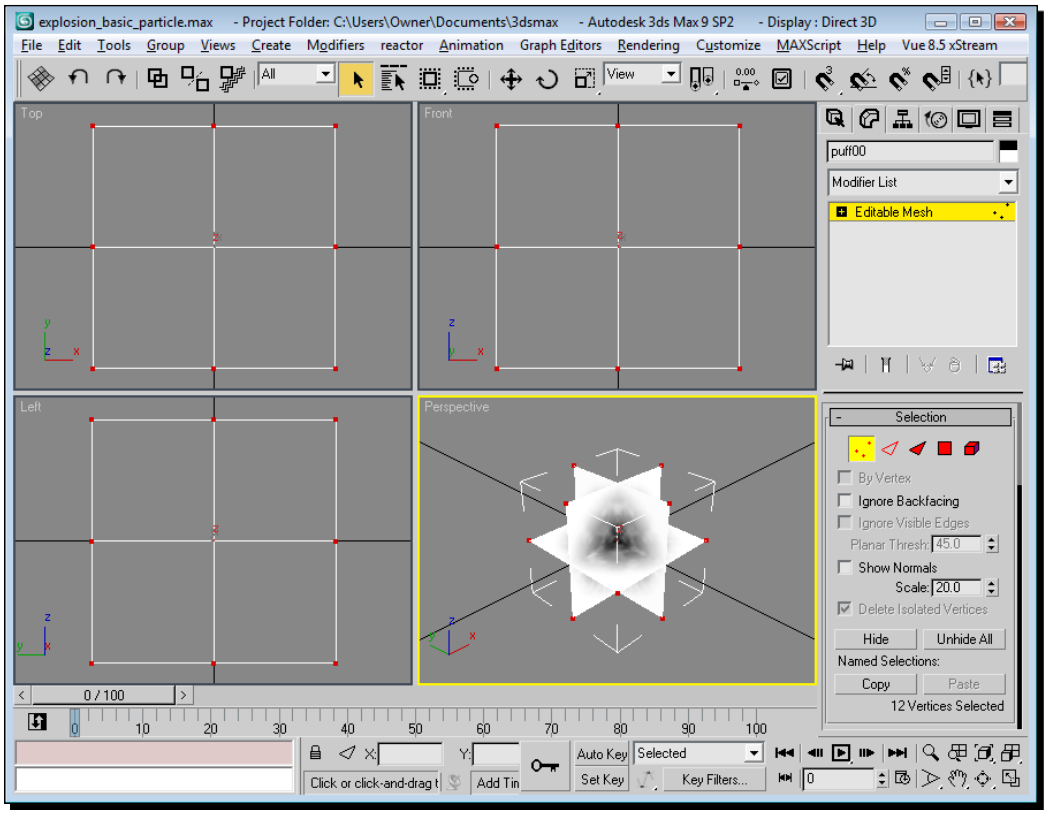
As the purpose of this book is to focus on the programming side of the game design, the art creation work is up to you. Here are four textures that we could use in this example. These were made using Photoshop and saved as `.PNG` files with transparency. As we are going to render with an additive (whitening) blendmode, we can save them as small (128x128) `.JPG` images with a black background in order to keep the file size of your SWF minuscule:



Time for action – sculpting a single particle

Now that we have some lovely looking textures, let's get them rendered in 3D. We don't want to waste CPU by animating each and every sprite individually in our game, so instead we want to "sculpt" some nice looking "blobs" comprised of multiple polygons. Fire up your favorite 3D modeling application such as 3D Studio Max, Blender, or Maya, and create a still version of a few kinds of special effect that you wish to use.

As a case study, we will go through the process of making a frequently used example: an explosion. Start by creating a flat quad plane and texture it. This is a single particle.

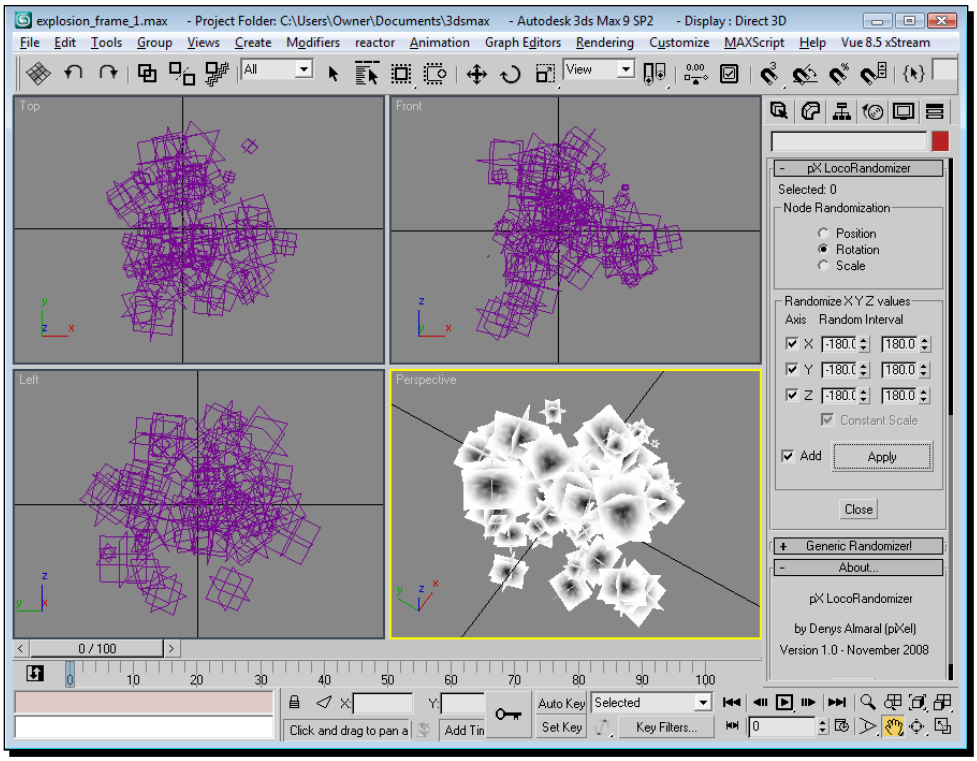


Time for action – sculpting a group of particles

In a typical explosion or water splash or group of sparks, we generally expect more than one. Clone this plane several times until you have a cluster of particles that looks close to the real thing.

One optional technique you may wish to use (depending on the shape and style you are hoping to achieve) is to create a three-quad base mesh and clone it rather than use each plane individually. As a single quad viewed from certain angles will have no thickness, a great technique is to use three quads, each intersecting at perpendicular angles, so that there is one facing each axis (x, y, and z). In other words, make little "stars" that can be viewed from any camera angle.

Group those together and start cloning this shape many times for a truly great looking particle cloud. Scatter it about randomly—perhaps more closely packed together in the center of the cluster, with smaller variants near the edges.



When you are happy with the look of your explosion, export it as a triangulated `.OBJ` mesh.

Time for action – sculpting the second keyframe

Now start working on the second frame of your particle animation. This is a "keyframe" that will be used during the interpolated animation in your game. Nudge each particle slightly upward, scale each one a little larger, and rotate it a tiny bit. There are macros that can do this randomly for multiple objects at the same time. Save this as a second `.OBJ` file, making sure that your two meshes have the exact same number of polies, each in the same order.

What just happened?

In the preceding steps, we briefly outlined a way to create two meshes for use as the before and after state for our particle system. The `Stage3dParticle` class takes these two meshes and morphs the first into the second over time by using a custom shader.

Repeat this process with all the special effects you wish to render in your game. You can really have a lot of fun with this step. Use your imagination and come up with other interesting shapes. For example, you might sculpt a circular or ring-shaped shock wave. Another frequently used particle shape is a small cluster of "shooting stars"—a cluster of particles that points in a certain direction like some sort of directional spray. This mesh could be used for splashes of water, lava, blood, oil, or sparks. Conical shapes with clusters of particles pointing in one direction spreading outward are also perfect for engine glows, blowtorches, and muzzle-flashes from guns.

As this book is about the programming, we won't go into the art any further, but this step is a great place to go crazy and let your imagination run wild. Stage3D is so fast that you don't really have to worry about how many polygons you are using. That said, the fewer you use the better the framerate, so try to create meshes in the under 1,000 polygon range. You can often create beautiful clusters of particles using far fewer than that.

Incorporating the particle system class in our game

The final step in our quest to deliver gratuitous amounts of glorious eye-candy is to get it up and running inside our game. This step is very simple and will only require a few extra lines of code and a couple changes here and there.

Time for action – adding particles to your game

Start with the example project from the previous chapter and open your `Stage3dGame.as` file. Import our new classes by adding the following lines at the top of your file near all the other import statements:

```
import Stage3dParticle;
import GameParticlesystem;
```

Next, add a few more variables inside the class definition, just below the Vector lists of enemies, bullets, and props:

```
// used for our particle system demo
private var nextShootTime:uint = 0;
private var shootDelay:uint = 0;
private var explo:Stage3dParticle;
private var particleSystem:GameParticlesystem;
private var scenePolycount:uint = 0;
```

Now embed the new meshes you have created along with any new textures in the same fashion as is already used for our spaceship, asteroids, and sky models:

```
// explosion start - 336 polygons
[Embed (source = "explosion1.obj",
  mimeType = "application/octet-stream")]
private var explosion1Data:Class;

// explosion end - 336 polygons
[Embed (source = "explosion2.obj",
  mimeType = "application/octet-stream")]
private var explosion2Data:Class;
```

Do the same for any other particle meshes you have designed.

Make sure to embed all new textures you created in the same way as the others, using either `[embed]` or the Flash IDE library. See the previous chapter source for examples.

Time for action – preparing a type of particle for use

Now that your new art content has been embedded, ensure that it is parsed—again in the same way as the existing models and textures in our game. Specifically, add any new textures to your `onContext3DCreate()` function by copy-n-pasting the working code used by our other textures.

```
particle1texture = context3D.createTexture(
    particle1bitmap.width, particle1bitmap.height,
    Context3DTextureFormat.BGRA, false);
  uploadTextureWithMipmaps(
    particle1texture, particle1bitmap.bitmapData);
```

In order to define this new particle type, add the following lines to the very bottom of your `initData()` function:

```
// create a particle system
particleSystem = new GameParticlesystem();

// define an explosion particle type
particleSystem.defineParticle("explosion",
  new Stage3dParticle(explosion1Data, context3D,
  particle1texture, explosion2Data));
```

What just happened?

The preceding code first creates a new `GameParticlesystem` object that will, from now on, handle all the special effects in our game. We then create a new `Stage3dParticle` object by sending both frames of mesh data along with an already initialized texture and Context3D. This is the "master clone source" version that will be duplicated on demand by our particle system. We give this type of particle a descriptive name ("explosion") and instruct it what to clone each time we want to spawn a new particle.

In the example game from this chapter, there are five different kinds of particle defined in the same way as illustrated earlier. For simplicity, it is left up to the reader to implement different kinds of particle. Use your imagination!

Time for action – upgrading the renderScene function

We need to insert code in our render loop to draw all the particles. As an added bonus, let's keep track of the number of polygons being rendered in each scene. A few additional lines are required to add them together. This is what your final `renderScene` function should look like:

```
private function renderScene():void
{
  scenePolycount = 0;

  viewmatrix.identity();
  // look at the player
  viewmatrix.append(chaseCamera.transform);
  viewmatrix.invert();
  // tilt down a little
  viewmatrix.appendRotation(15, Vector3D.X_AXIS);
  // if mouselook is on:
  viewmatrix.appendRotation(gameinput.cameraAngleX,
    Vector3D.X_AXIS);
  viewmatrix.appendRotation(gameinput.cameraAngleY,
    Vector3D.Y_AXIS);
  viewmatrix.appendRotation(gameinput.cameraAngleZ,
    Vector3D.Z_AXIS);

  // render the player mesh from the current camera angle
  player.render(viewmatrix, projectionmatrix);
  scenePolycount += player.polycount;

  // loop through all known entities and render them
  for each (entity in props)
```

```
    {
      entity.render(viewmatrix, projectionmatrix);
      scenePolycount += entity.polycount;
    }

    particleSystem.render(viewmatrix, projectionmatrix);
    scenePolycount += particleSystem.totalpolycount;
  }
```

What just happened?

The only changes since the last time are the addition of the polycount stats and the `particleSystem.render` call. Remember that the `particleSystemrender()` function will loop through each and every particle and draw it for us. We simply pass the view and projection matrix so the camera angles are respected and let it do its work.

Time for action – adding particles to the gameStep function

Now that we have set things up, let's experiment with particles, so that when you hold down the spacebar, more and more particles appear on the screen. Edit the `gameStep()` function such that a new particle is spawned in each frame when the "fire" input is pressed. Additionally, we need to instruct the particle system to update each frame based on the elapsed time. Here is the new version of this function:

```
  private function gameStep(frameMs:uint):void
  {
    // handle player input
    var moveAmount:Number = moveSpeed * frameMs;
    if (gameinput.pressing.up) player.z -= moveAmount;
    if (gameinput.pressing.down) player.z += moveAmount;
    if (gameinput.pressing.left) player.x -= moveAmount;
    if (gameinput.pressing.right) player.x += moveAmount;
    if (gameinput.pressing.fire)
    {
      if (gametimer.gameElapsedTime >= nextShootTime)
      {
        //trace("Fire!");
        nextShootTime =
          gametimer.gameElapsedTime + shootDelay;
        // random location somewhere ahead of player
        var groundzero:Matrix3D = new Matrix3D;
        groundzero.prependTranslation(
          player.x + Math.random() * 200 - 100,
          player.y + Math.random() * 100 - 50,
```

```
            player.z + Math.random() * -1000 - 250);
          // create a new particle (or reuse an inactive one)
          particleSystem.spawn("explosion", groundzero, 2000);
        }
      }
      // follow the player
      chaseCamera.x = player.x;
      chaseCamera.y = player.y + 1.5; // above
      chaseCamera.z = player.z + 3; // behind
      // animate the asteroids
      asteroids1.rotationDegreesX += asteroidRotationSpeed * frameMs;
      asteroids2.rotationDegreesX -= asteroidRotationSpeed * frameMs;
      asteroids3.rotationDegreesX += asteroidRotationSpeed * frameMs;
      asteroids4.rotationDegreesX -= asteroidRotationSpeed * frameMs;
      // animate the engine glow - spin fast and pulsate slowly
      engineGlow.rotationDegreesZ += 10 * frameMs;
      engineGlow.scaleXYZ =
        Math.cos(gametimer.gameElapsedTime / 66) / 20 + 0.5;
      // advance all particles based on time
      particleSystem.step(frameMs);
    }
```

What just happened?

The preceding code steps the particle system (by passing the number of milliseconds that have elapsed since the previous frame) and also spawns a new explosion at a random location that is set to last for 2,000 milliseconds (or two seconds) if the spacebar is pressed. As this function is called every frame, if you hold down the spacebar a massive number of particles will be spawned.

Time for action – keeping track of particle statistics

As we are getting to the point that screenshots and boastful stats become fun to share with friends, a little extra information can go a long way to prove the efficiency of our particle routines.

Firstly, in our `heartbeat` function (which is only run every few seconds and at the moment is only used for debug purposes), let's report some more statistics. We do this as follows:

```
// for efficiency, this function only runs occasionally
// ideal for calculations that don't need to be run every frame
private function heartbeat():void
{
  trace('heartbeat at ' + gametimer.gameElapsedTime + 'ms');
```

```
    trace('player ' + player.posString());
    trace('camera ' + chaseCamera.posString());
    trace('particles active: ' + particleSystem.particlesActive);
    trace('particles total: ' + particleSystem.particlesCreated);
    trace('particles polies: ' + particleSystem.totalpolycount);
  }
```

Finally, along with the framerate, we will report how many polies are being rendered so our screenshots can have something to boast about. Edit the one line in our `enterFrame` function that updates the FPS text and add the `scenePolycount` as follows:

```
fpsTf.text = fps.toFixed(1) +
      " fps (" + scenePolycount + " polies)";
```

What just happened?

We upgraded our game to keep track of the number of polygons being rendered in each frame. This will help during testing to give benchmarks over time in the debug log (to determine if the game is becoming too filled with enemies or particles after long play sessions) and so people can compare the framerate with the number of polies on the screen at any given moment.

As many popular particle systems currently get bogged down with only a few thousand particles (because they are simulating on the CPU and render each particle in a separate draw call), this "batched GPU simulated" particle system is sure to impress your friends with its incredible performance. You can expect to be able to animate 50,000+ particles per frame and still maintain a smooth 60fps on decent gaming rigs.

That is it! Our game can now pump out massive numbers of particles of various kinds. The preceding example code only adds the "explosion" particle type, but in your own game you can do the same for several types of particle.

Let's see the new particle system in action!

You can view the finished *Chapter 8* demo here:

`http://www.mcfunkypants.com/molehill/chapter_8_demo/`

In addition, the full source code is available at the following URL:

`http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book`

Now that we have implemented a fancy new particle entity class, AGAL shader code to do all particle animation on the GPU for speed and a particle system that maintains a pool of reusable particles, our game is capable of some incredible eye-candy. As we are batching geometry and rendering hundreds or even thousands of particles in a single `drawTriangles` call, our game can easily render tens of thousands of particles on the screen without even breaking a sweat.

Rendering GPU particles using the batched, reusable, hyper-optimized technique outlined in this chapter frees up your CPU for other things. This means that your game has lots of horsepower left to render other kinds of meshes in your level, as well as the CPU being available to process inputs, trigger sounds, perform AI and collision detection, and more without having to do any of the heavy lifting required for effects.

Pop quiz – earn some experience points

Here are a few quick questions to test your understanding. If you can answer them, you are officially ready to move on to the next step in your grand adventure:

1. Why could we not just render thousands of individual particles separately instead of in large batches?
    a. Because Stage3D can only display low-polygon scenes
    b. Because each draw call has overhead and too many will result in poor framerate
    c. Because we would have to write a unique shader for each one
    d. Because it would be too boring for our computer
2. Why did we create two meshes for our explosion instead of just one?
    a. As a means to pre-calculate the trajectories of numerous polygons
    b. So that we don't have to use the CPU to do any simulation work
    c. Because we use the second to smoothly animate particles in an AGAL shader
    d. All of the above
3. What technique do we use to increase the framerate and to avoid memory leaks and GC (garbage collection) of temporary variables during our render loop?
    a. We made all our variables private
    b. We assumed all users have infinite RAM
    c. We made a "pool" of reusable particles that can be respawned over and over
    d. We compiled our source code very carefully

Have a go hero –inventing some cool particle effects

Your side quest this time is to design a bunch of cool-looking particle effects for your game. Instead of just a simple expanding explosion, why not add a swirling tornado (made up of many dust motes or hazy cloud sprites)? Alternatively, create a texture made up of small blue droplets and design a "splash" appropriate for the front of a sailing ship on the ocean. Perhaps your game will be a shooter and you want gratuitous gore. If so, make blood splatters that emerge from gunshot wounds along with glowing muzzle flashes coming from your guns.

Particles can also be used in other less obvious ways. For example, in a detective game, faint footprints in the sand could be spawned every few feet of movement by players. Debris such as litter and paper in a ghetto blowing in the wind can add atmosphere to a game. In an underwater game, small specks of organic material floating in the water can give your movement more depth. In a dating simulator or visual novel, perhaps a character is so enamored with another that little red hearts float upward from their eyes as they fall madly in love.

Summary

In this chapter, we managed to achieve the following milestones:

- We wrote an AGAL shader that smoothly animates and fades out particles over time
- We designed a basic particle class with two-frame keyframed mesh data
- We created a particle system that manages and reuses multiple types of particle
- We upgraded our game to show off these new special effects

Level 8 achieved!

Congratulations. You are nearly at the end of your epic journey. Though our game is still very simple, we have achieved a major milestone along the way towards the creation of a high-poly, next-gen, amazing-looking 3D game in Flash. On the horizon are all the final changes that make a video game. These last pieces in the puzzle represent the exciting conclusion to our grand adventure.

There is one last quest to complete. The tying up of loose ends: the polish. This is the challenge that awaits us in the next two chapters. The treasure that we get as a reward, after all our hard work is that we will soon be able to play a complete 3D game of our own design.

Where to buy this book

You can buy Adobe Flash 11 Stage3D (Molehill) Game Programming Beginner's Guide from the Packt Publishing website: `http://www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/adobe-flash11-stage3d-molehill-game-programming-beginners-guide/book