

# Optimization Tutorial: Particles and High-Frequency Code

## Overview

Particle systems are just cool. Even the simplest can generate fascinating patterns, and they are enormous fun to watch. In an effort to better understand the performance characteristics of the XNA Framework with respect to Windows and the Xbox 360, I built a simple particle system. The results were interesting and show that some simple optimizations can result in dramatic improvements to performance on both the Xbox 360 and Windows.

The goals of this white paper are twofold. First, bring to light some of the interesting performance issues encountered when programming the Xbox 360 using the XNA Framework. Second, and more important, reveal how you go about finding such performance issues and correcting them without the use of advanced profiling tools.

## Starting Out

I started by creating a list of 10,000 particles and randomly spreading them across an 800x600 field. The particles are rendered using a `DrawUserPrimitives` call with a simple screen-space shader. This is easy to replicate here. First, create a Windows Game (2.0) project in Visual Studio, and then add the following members.

```
VertexDeclaration particleDeclaration;

const int numParticles = 10000;
VertexPositionColor[] particles;
Vector4 white = new Vector4(1.0f, 1.0f, 1.0f, 0.3f);
Effect screenShader;
```

And initialize these in the `Initialize` function provided by the framework. In this case, I created an `InitParticles` function, which is called by `Initialize`.

```
protected void InitParticles()
{
    particles = new VertexPositionColor[numParticles];
    int backBufferWidth =
        graphics.GraphicsDevice.PresentationParameters.BackBufferWidth;
    int backBufferHeight =
        graphics.GraphicsDevice.PresentationParameters.BackBufferHeight;
    Random random = new Random();
    for (int i = 0; i < numParticles; i++)
    {
        particles[i].Position.X = random.Next(backBufferWidth);
        particles[i].Position.Y = random.Next(backBufferHeight);
        particles[i].Position.Z = 0.5f;

        particles[i].Color = new Color(white);
    }
}
```

Given that this is a two-dimensional particle system, why are we using a `Vector3` to store the particle positions? We want to use the draw call `DrawUserPrimitives` for rendering the entire array of particles that we have. It's much easier to use a defined vertex type (in this case,

VertexPositionColor) with this call, rather than go through all of the hoops of creating our own vertex type and the predefined vertex type uses a `Vector3`. This does lead to some inefficiency later on, but the `Vector3` is not the cause of the performance issues, as will become evident.

Anytime you create a resource on the `GraphicsDevice`, you should do it in the `LoadContent` function provided by the framework. Inside this function, I create the `VertexDeclaration` we need. We also load and create the screen space shader we need. This assumes the shader is named `simplescreen.fx` and resides in the `content\shaders` directory.

```
// TODO: Load any ResourceManagementMode.Automatic content
screenShader = Content.Load<Effect>(@"shaders\simplescreen");
particleDeclaration = new VertexDeclaration(
    graphics.GraphicsDevice, VertexPositionColor.VertexElements );
```

We want the particle to move, eventually, but let's just get them drawing first. It always feels good to get something rendering. In the `Draw` function provided by the framework, we change the `CornflowerBlue` to `Black` to more easily see the white particles, and then we add the following code to draw them.

```
graphics.GraphicsDevice.Clear(Color.Black);

// Draw using PointList
graphics.GraphicsDevice.VertexDeclaration = particleDeclaration;
screenShader.Begin();
{
    EffectPass pass = screenShader.CurrentTechnique.Passes[0];
    pass.Begin();
    graphics.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
        PrimitiveType.PointList, particles, 0, numParticles);
    pass.End();
}
screenShader.End();
```

We now need the screen space shader `.fx` file. Create a new folder under project called `Content` and create a new folder under `Content` called `Shaders`. Right-click on the `Shaders` folder, click **Add**, and then click **New Item** from the context menu. Select **Text File** and name it `SimpleScreen.fx`. Open it in the editor by double-clicking the entry, and add this code to it.

```
struct VS_INPUT
{
    float4 ObjectPos: POSITION;
    float4 VertexColor: COLOR;
};

struct VS_OUTPUT
{
    float4 ScreenPos: POSITION;
    float4 VertexColor: COLOR;
};

float halfScreenWidth;
float halfScreenHeight;
```

```

VS_OUTPUT SimpleScreenVS(VS_INPUT In)
{
    VS_OUTPUT Out;

    //Move to screen space.
    Out.ScreenPos.x = (In.ObjectPos.x - halfScreenWidth) / halfScreenWidth;
    Out.ScreenPos.y = (In.ObjectPos.y - halfScreenHeight) / halfScreenHeight;
    Out.ScreenPos.z = 0;
    Out.ScreenPos.w = 1;

    Out.VertexColor = In.VertexColor;

    return Out;
}

float4 SimpleScreenPS(float4 color : COLOR0) : COLOR0
{
    return color;
}

//-----//
// Technique Section for Simple screen transform
//-----//
technique SimpleScreen
{
    pass Single_Pass
    {
        CULLMODE = NONE;
        POINTSIZE = 1;

        VertexShader = compile vs_1_1 SimpleScreenVS();
        PixelShader = compile ps_1_1 SimpleScreenPS();
    }
}

```

This shader simply takes the position passed in, which is in the range from 0 to `screenWidth` wide by 0 to `screenHeight` high and calculates where that goes on screen. We simply output the color stored in the vertex in the pixel shader and we're done.

This means we now need to set the `halfScreenWidth` and `halfScreenHeight` variables for the shader in the C# code. We do this by adding these lines to the `LoadContent` function after the shader gets loaded.

```

screenShader = Content.Load<Effect>(@"shaders\simplescreen");
halfScreenHeight = screenShader.Parameters["halfScreenHeight"];
halfScreenWidth = screenShader.Parameters["halfScreenWidth"];

```

This means we need to define these parameters as well in the class. You can do this by adding these lines to declarations section at the top of the main class.

```

EffectParameter halfScreenWidth;
EffectParameter halfScreenHeight;
Effect screenShader;

```

This also means we need to set them to the correct values or the screen will be sadly blank. Right after the `Clear` statement in the `Draw` function, we need to add the following:

```
graphics.GraphicsDevice.Clear(Color.Black);

halfScreenHeight.SetValue(graphics.GraphicsDevice.PresentationParameters.Back
BufferHeight / 2);

halfScreenWidth.SetValue(graphics.GraphicsDevice.PresentationParameters.Back
BufferWidth / 2);
```

If you've typed this correctly, you should see an 800x600 window with a random field of white particles filling it. Cool, but we're really just scratching the surface here.

### Make 'Em Move

The next step is to get them moving. To do that, we will need to add a couple of things. First, we need to add the attractor. This will be a point at the center of the screen that attracts the particles toward it. The particles interact only with the attractor and not each other. If the particles are going to move, we need to keep track of their velocities and update them every frame with the acceleration caused by the attractor. To keep things simple, we'll treat the attractor as a linear force that varies based on distance from the attractor. First, let's add the new members we need:

```
//Simulation
Vector3 attractorPosition;
const float attractorForce = 1000.0f;
Vector3[] particleVelocity;
```

These members are all part of the simulation and will be changed during the `Update` function provided by the framework. Let's do the initialization we need. The particle velocities are created in the `InitParticles` function like this:

```
particles = new VertexPositionColor[numParticles];
particleVelocity = new Vector3[numParticles];
int backBufferWidth =
    graphics.GraphicsDevice.PresentationParameters.BackBufferWidth;
int backBufferHeight =
    graphics.GraphicsDevice.PresentationParameters.BackBufferHeight;
Random random = new Random();

for (int i = 0; i < numParticles; i++)
{
    particles[i].Position.X = random.Next(backBufferWidth);
    particles[i].Position.Y = random.Next(backBufferHeight);
    particles[i].Position.Z = 0.5f;

    particles[i].Color = new Color(white);

    particleVelocity[i] = Vector3.Zero;
}
```

We then create an `InitAttractor` function to initialize the attractor data. Again, we place the attractor in the center of the screen. Be sure to call this in the `Initialize` function!

```
protected void InitAttractor()
{
    attractorPosition.X =
        graphics.GraphicsDevice.PresentationParameters.BackBufferWidth / 2;
    attractorPosition.Y =
        graphics.GraphicsDevice.PresentationParameters.BackBufferHeight / 2;
    attractorPosition.Z = 0.5f;
}
```

Again, we are using a `Vector3` for velocity when the only velocities we will have are 2D. Why? Mostly, this is a consistency thing. If the positions are 3D, the velocities ought to be 3D as well. It also aids in constructing the code since the operators for `Vector3`'s really want to work with other `Vector3`'s. This also allows us to generalize the code from a 2D to a 3D particle system without major refactoring.

Now, let's add the update code. The framework provides a function that gets called every frame of the game and passes in the time elapsed since the last frame. Right underneath the `TODO`: we add:

```
// Update Time - Time is in fractions of a second
float currentTime = (float)gameTime.ElapsedGameTime.Milliseconds / 1000.0f;

// Update Particles - loop through each particle
for (int i = 0; i < numParticles; i++)
{
    Vector3 accelVector;
    // We need the direction to the attractor and distance from the particle
    we're checking
    accelVector = attractorPosition - particles[i].Position;

    float dist = accelVector.Length();
    float accel = 0.0f;

    // If we are within 1 pixel of the attractor, just turn force off to
    avoid infinities at the singularity!
    // Particle will speed through and out the other side
    if (dist > 1.0f)
    {
        accel = attractorForce / dist;
        accelVector /= dist;
    }

    // Calculate acceleration for this frame
    accelVector *= accel * currentTime;

    // Apply acceleration to the velocity
    particleVelocity[i] += accelVector;

    // Apply velocity to particle position
    particles[i].Position += particleVelocity[i] * currentTime;
}
```

That's it! Let's give this a spin... When you run this, the particles should stream through the center of the screen and out the other side.

## Simple Edge Collision

In fact, they probably pick up enough speed to head completely offscreen, which makes the scene rather sparse pretty quickly. Some of them head back in, and already the pattern is very cool! Let's have them bounce back when they hit the edge of the screen. This is really easy to do. Simply add the following code to the Update function in the particles loop right before the final brace of the loop:

```
if ((particles[i].Position.X < 0.0f) ||
    (particles[i].Position.X >
     graphics.GraphicsDevice.PresentationParameters.BackBufferWidth))
    particleVelocity[i].X = -particleVelocity[i].X;
else if ((particles[i].Position.Y < 0.0f) ||
         (particles[i].Position.Y >
          graphics.GraphicsDevice.PresentationParameters.BackBufferHeight))
    particleVelocity[i].Y = -particleVelocity[i].Y;
```

If we hit the edge of screen by going less than 0 or by going greater than the back-buffer width or height, simply reverse the velocity component of the direction we exceeded. Very simple and works like a champ. Let's give this a spin.

## Moving the Attractor

This is cool but will generate the exact same thing every time. Not the most fun in the world after about the third time you run it. Let's give the attractor an initial velocity and update its position each frame. This should pull the particles in the world along with it.

First, let's add in the data member we need:

```
Vector3 attractorVelocity;
```

Now, initialize the velocity to a random value in the InitAttractor function. This code comes right after the position initialization code.

```
Random random = new Random();

attractorVelocity.X = random.Next(150) - 75;
attractorVelocity.Y = random.Next(150) - 75;
attractorVelocity.Z = 0.0f;
```

All we need now is update the attractor position each frame in the Update function. Add this code between the // Update Time code and the //Update Particles code.

```
// Update Attractor Position
attractorPosition += attractorVelocity * currentTime;

// Attractor reflects off the screen edges too or it just disappears!
if ((attractorPosition.X < 0.0f) ||
    (attractorPosition.X >
     graphics.GraphicsDevice.PresentationParameters.BackBufferWidth))
    attractorVelocity.X = -attractorVelocity.X;
else if ((attractorPosition.Y < 0.0f) ||
         (attractorPosition.Y >
          graphics.GraphicsDevice.PresentationParameters.BackBufferHeight))
    attractorVelocity.Y = -attractorVelocity.Y;
```

Running this should get you an attractor that moves differently every time and generates really cool patterns that are different every time.

## Final Touches

There are two problems with this implementation. First, the attractor is invisible and hard to track, although its effects are pretty obvious. Second, after a minute or two, the particles are moving so fast that all patterns have disappeared and you just have noise.

To fix the first problem, we will use some shader magic and draw the particles alpha blended. This will make the patterns easier to see. We will draw the particles closest to the attractor as opaque, which will make it obvious where the attractor is. You want to do these color calculations in the shader because shaders are good at this sort of work, and it lessens the burden on the CPU. The first step is letting the shader know where the attractor is. We do this by creating a shader parameter we can pass into the shader from the C# code. Let's add this line to shader right after the `struct VS_OUTPUT` definition:

```
float3 attractorPos;
```

In order to pass this parameter to the shader, we need to add the following lines to the C# program. First, we need to define the `EffectParameter` member to the `Game` class. Right after the `Color` declaration for `white` at the top, add this line:

```
EffectParameter attractorPos;
```

Next, we define how the `EffectParameter` maps to the shader parameter. In the `LoadContent` function, right after the `.Load` call for the `Effect`, we add this line:

```
attractorPos = screenShader.Parameters["attractorPos"];
```

This tells the `EffectParameter` to map whatever value I give it in the C# code to the `attractorPos` variable we just defined in the shader. Now, there's only one thing left to do in the C# code and that is to put the value into the shader parameter each time we call `Draw`. In the `Draw` function, add the following line, right after the `Clear` call:

```
attractorPos.SetValue(attractorPosition);
```

Now, we have to add the code to the shader that actually does the work. This turns out to be pretty simple:

```
//Calculate alpha for point based on distance from attractor
float3 distV = attractorPos - In.ObjectPos;
float dist = 5.0f / sqrt((distV.x * distV.x) + (distV.y * distV.y));

float alpha = 0.3f + 0.7 * dist;

Out.VertexColor = In.VertexColor;
Out.VertexColor.a = alpha;
```

We simply calculate the alpha value based on distance from the attractor. The closer it is, the more opaque it becomes, and it's never allowed to become more transparent than 0.3.

There is one other thing we need to do, which is to tell the Effect file to set more render states. This is because the default behavior in the XNA Framework is to have alpha blending off. To turn it on, we simply add the render states we need to the `Pass` section of the shader:

```
pass Single_Pass
{
    CULLMODE = NONE;
    POINTSIZE = 1;
    ALPHABLENDENABLE = TRUE;
    SRCBLEND = SRCALPHA;
    DESTBLEND = INVSRCALPHA;
    ZENABLE = FALSE; //Always want this on top

    VertexShader = compile vs_1_1 SimpleScreenVS();
    PixelShader = compile ps_1_1 SimpleScreenPS();
}
```

To fix the second problem, we need to introduce a drag force on the particles that slows them down in direct proportion to their velocity. We can do this with a single line of code. Just add the following line to the particle update code, right after the line to apply acceleration to the velocity.

```
particleVelocity[i] -= particleVelocity[i] * dragForce * currentTime;
```

Of course, we do need to create the `dragForce` and initialize it as well. Back up top, under the `attractorForce` we add:

```
const float dragForce = 0.01f;
```

At this point, we have a pretty good implementation on Windows. Frame rate for me on Core2 Duo CPUs was pretty smooth, utilizing about 50-odd percent of one core with the cores running at 2.00 GHz in Windows Vista.

## A Rude Awakening

The next step gets it running on Xbox 360. We've all heard about the power of the next-generation consoles through innumerable ads, game commercials, and web sites. It should just be a question of running the code on the console and calling it a day, right? This turns out not to be the case.

First, we need to move this code over to an Xbox 360 game project. This is easy to do in XNA Game Studio 2.0.

Open the Particles project in Visual Studio. In the Solution Explorer window, right-click on the project entry and select "Create Copy of Project for Xbox 360." This will create a copy of the Windows project in the same solution, referencing the same code and content. Right-click again on the new project entry and select "Rename," and change the name to "Particles360." Finally, right-click on the new project entry again and select "Set as StartUp Project".

Assuming your Xbox 360 is running XNA Game Studio Connect, and you have established a connection to your PC with the XNA Game Studio Device Center, press the run button in Visual Studio C# Express. It

should build and deploy to the Xbox 360 and then start running. Yowza!!! That is bad frame rate! Where is this touted next-gen performance? What have I done wrong?

Before you cancel your subscription to the Creators Club, let's look at the code and see what we can do...

### **Optimization 101 – Comment stuff out and see if it helps...**

First off, you may have noticed that there aren't many tools to help with this at present. That's OK! There didn't used to be a lot of tools to help any performance analysis. We'll fall back on the tried and true, and you'll be amazed at how well this works.

Some knowledge of the Xbox 360 hardware would be good at this point. The Xbox 360 has 512 MB of RAM, three PowerPC cores with two hardware threads on each running at 3.2 GHz and a GPU that is as fast as any existing DirectX 9 part today. What does information tell us? Well, if you're running OK on Windows and you don't have the highest-end DirectX 9 part installed on your computer, the GPU on the Xbox 360 probably isn't the problem. Even if you have the highest-end DX9 part available, so does the Xbox 360! This means much of the code in `Draw` can probably be ignored. Let's make that assumption and focus on the `Update` function.

There is actually is some pretty serious CPU code here: Big loop, fair amount of math per particle, lots of particles. This ran fine on Windows, so why would the Xbox CPU, which is running at 1.2 GHz *faster* than my Windows PC (half-again the speed) have trouble with this? Your first instinct will be to blame some aspect of managed code; the JITer, the Garbage Collector, blah, blah, blah. And in this case, you would be right, but it is the compiler in combination with the Xbox 360 CPU that has betrayed us.

See, the Xbox 360 CPU cores have a couple of interesting problems associated with them. First, they have no out-of-order instruction fixup. This means that if the instructions arrive at the CPU in a way that isn't optimal, the CPU generates stalls to keep things in sync. This is in direct contrast to the Intel and AMD CPUs for Windows, which have a fair amount of logic to reorder instructions as they come in to help the CPU digest them. The Xbox 360 CPUs are also RISC-based processors. A complex instruction set processor like the Intel and AMD processors have many, many complex, essentially single-step instructions to do what will take 2 to 5 instructions to perform on a RISC processor. Since we are compiling this to MSIL and then Just-In-Time compiling it on the Xbox 360, the compiler doesn't have a lot of opportunity to reorder instructions the way the C++ compiler can. It also doesn't get much opportunity to make use of the tremendous numbers of registers on the Xbox 360 to prevent load and storing from memory to the actual operations. Since the C++ compiler is an offline compiler, it can crunch on the code and help the Xbox 360 processor by reordering instructions that would cause these stalls and also be "register aware" to prevent loads and stores. The MSIL compiler won't do this because it is processor agnostic, and the JITer is blissfully unaware of the CPU issues on the Xbox 360. The JITer must brute force everything.

If we look at the `Update` function, we see one section of code that runs once every frame and another section that executes once for each particle every frame. We should focus our attention on the loop, since any optimization we make in the loop gets multiplied by the total number of particles we are updating.

So what can we do? Let's comment out the following in `Update`. Let's not drag the particles, and let's not let them collide with the screen edge. The particles will still move, will not slow down, and will not stay on screen. Most of the particle logic will still run.

```
/*
// Apply the drag force to particle velocity
particleVelocity[i] -= particleVelocity[i] * dragForce * currentTime;

if ((particles[i].Position.X < 0.0f) ||
    (particles[i].Position.X >
     graphics.GraphicsDevice.PresentationParameters.BackBufferWidth))
    particleVelocity[i].X = -particleVelocity[i].X;
else if ((particles[i].Position.Y < 0.0f) ||
         (particles[i].Position.Y >
          graphics.GraphicsDevice.PresentationParameters.BackBufferHeight))
    particleVelocity[i].Y = -particleVelocity[i].Y;
*/
```

What happens? Well, we get a compiler warning and the Xbox 360 runs MUCH faster. Indeed, feels just fine! How is this possible? My opinion of the Xbox 360 hasn't improved much based on this!

### Optimization 102 – Why did this make it faster?

The key is to think about the compiler and the processor. The compiler can't reorder instructions, but it also cannot inline functions. If we look at the one line of math we are doing above:

```
particleVelocity[i] -= particleVelocity[i] * dragForce * currentTime;
```

What this is doing, based on order of operations, is using the **Vector3** multiply operator override for a scalar and multiplying `particleVelocity[i] * dragForce`. This is a function call (remember, we can't inline), and the operator multiplies each member of the vector (X, Y, and Z) by `dragForce`. It then takes the result of that operation and calls the **Vector3** multiply operator override for a scalar and multiplies by the current time. Since all of the operations are multiply, the order of operations is from left to right. By grouping the scalar multiply, we save the compiler from doing two multiplies!

```
particleVelocity[i] -= particleVelocity[i] * (dragForce * currentTime);
```

This also saves us calling one of the operator overloads. This kind of thing is frequently fixed up by modern C++ compilers. They inline aggressively, and a tremendous amount of work has been done to improve math code using the built in floating-point units in Intel/AMD chips, including keeping things on the floating-point unit stack until we really need to save them off. It is not fixed by managed code compilers for a number of reasons. Probably the simplest is they simply aren't that mature yet. C++ compilers have been around for decades. Managed code compilers have not. Is this enough to fix the problem? Let's try it and see. Comment in the drag math line and just group the scalars like the line above.

This seems to run just fine. And now we have drag at a decent frame rate. But we still have an unanswered question. Was it the extra multiplies or the operator overload call that slowed us down? Let's try and find out. Let's remove the operator overloads. Change the single line of drag math to:

```
particleVelocity[i].X -= particleVelocity[i].X * dragForce * currentTime;
```

```
particleVelocity[i].Y -= particleVelocity[i].Y * dragForce * currentTime;
particleVelocity[i].Z -= particleVelocity[i].Z * dragForce * currentTime;
```

No need to group anything anymore since the math operations are on scalar values. We could write this:

```
float dragTime = dragForce * currentTime;
particleVelocity[i].X -= particleVelocity[i].X * dragTime;
particleVelocity[i].Y -= particleVelocity[i].Y * dragTime;
particleVelocity[i].Z -= particleVelocity[i].Z * dragTime;
```

But we want to make sure we do ALL of the multiplies from the original line of code and just remove the operator overloads. When you are optimizing and tuning for performance, only change one condition at a time. If you don't, you don't know which condition or thing you changed is the real problem. Let's run this new bit...

It runs just fine! It wasn't the multiplication, since we are doing all of the multiplies that we were in the slow case. It must be the operator overloads. Note how the only tools we've needed so far are a basic understanding of the Xbox 360 CPU, our brains, and a text editor! And this is without even more stuff we could do. For example, we can remove two multiplies by using the pre-multiply above. We can also remove all multiplies involving the Z member. Why? Let's look at the shader code that consumes this set of vertices.

```
VS_OUTPUT SimpleScreenVS(VS_INPUT In)
{
    VS_OUTPUT Out;

    // Move to screen space. Assumes 800 x 600 viewport!
    Out.ScreenPos.x = (In.ObjectPos.x - 400) / 400;
    Out.ScreenPos.y = (In.ObjectPos.y - 300) / 300;
    Out.ScreenPos.z = 0;
    Out.ScreenPos.w = 1;

    // Calculate alpha for point based on distance from attractor
    float3 distV = attractorPos - In.ObjectPos;
    float dist = 5.0f / sqrt((distV.x * distV.x) + (distV.y * distV.y));

    float alpha = 0.3f + 0.7 * dist;

    Out.VertexColor = In.VertexColor;
    Out.VertexColor.a = alpha;

    return Out;
}
```

`In.ObjectPos` is the `Vector3` we are using the position of the particle. Note how there is no reference in the shader at all to `.z` component of this `Vector3`. We can safely leave it zero and never do any operation with it.

What about the other commented out lines? Does this mean I can't use collision with the edge of the screen on Xbox 360?

**Optimization 103 – The Secret Sauce™**

So, first off, let's comment in the collision lines and see what happens on the Xbox 360. Maybe we commented out too much to begin with, solved the only performance problem we had, and everything is now fine. Let's comment in the collision code.

```
if ((particles[i].Position.X < 0.0f) ||
    (particles[i].Position.X >
     graphics.GraphicsDevice.PresentationParameters.BackBufferWidth))
    particleVelocity[i].X = -particleVelocity[i].X;
else if ((particles[i].Position.Y < 0.0f) ||
         (particles[i].Position.Y >
          graphics.GraphicsDevice.PresentationParameters.BackBufferHeight))
    particleVelocity[i].Y = -particleVelocity[i].Y;
```

Ah, nope. This code is painful, too. Why? Some things that might go through one's head at this point:

- Managed code is really slow at conditional statements or at doing comparisons on Xbox 360. Must be OK on Windows since this doesn't happen there.
- Can't be operator overload this time 'cause we're using the scalar members of the `Vector3`.
- Probably isn't reversing the sign on a scalar since that doesn't happen very often. Only when a particle hits the edge of the screen, and clearly not a lot of them are doing this every frame.
- Maybe retrieving the data from the `GraphicsDevice` on Xbox 360 about the back buffer width and height is slow. Must be OK on Windows since this doesn't happen there.

Well, we've eliminated two of the four above. No operator overload this time. Changing the sign of the scalar isn't happening that often and probably not all at the beginning since everything is sucked into the center by the attractor, yet it still slows down.

This leaves us with the following:

- Conditionals and/or comparisons are slow in managed code. Yikes! I hope it's not that because it's hard to write any code without these.
- Retrieving data from the `GraphicsDevice` is slow on the Xbox 360. Hmm. It could be this, which is really easy to test.

How do we check this? Let's get the data from the `GraphicsDevice` for the width and height only once per `Update` call. That way, the screen resolution can change and we handle it. (Actually, the shader will fall on its face if we change screen resolution, but work with me here!) Change the collision block to:

```
if ((particles[i].Position.X < 0.0f) ||
    (particles[i].Position.X > backBufferWidth))
    particleVelocity[i].X = -particleVelocity[i].X;
else if ((particles[i].Position.Y < 0.0f) ||
         (particles[i].Position.Y > backBufferHeight))
    particleVelocity[i].Y = -particleVelocity[i].Y;
```

And let's set the `backBufferWidth` and `backBufferHeight` before the loop in `Update`:

```
// Get the backBuffer resolution just once
```

```
int backBufferHeight =
    graphics.GraphicsDevice.PresentationParameters.BackBufferHeight;
int backBufferWidth =
    graphics.GraphicsDevice.PresentationParameters.BackBufferWidth;

// Update Particles - loop through each particle
for (int i = 0; i < numParticles; i++)
```

Change the code to this and it looks like getting this data from the `GraphicsDevice` is slow on Xbox 360. This is great news because I don't have to come up with workarounds to comparison operations and conditional statements. But this does lead to another question. Why is it slow on Xbox 360? Herein lies the Secret Sauce™. Xbox 360 has a very different implementation of the Graphics Framework than Windows does. There are calls you can make on the Xbox 360 that incur a significant performance penalty over the same calls on Windows. The reasons for this would involve a digression that is worthy of a white paper on its own, so I will avoid that here.

Note that we didn't need the list to figure this out. This block of code clearly ran slow, and only a certain number of things could have caused it. It was easy to create a simple test that clearly showed the problem. And without any tools, again, but the editor and our brains!

### **If it helps Xbox 360, what about Windows?**

It's pretty easy to copy these changes into the Windows version and see what happens. If you are running on a processor built in the last 18 months or so, this almost certainly ran just fine for you without these optimizations. So how do I know it's any better? (sigh) Finally, we need a tool. And this one happens to be part of Windows XP or Windows Vista. Run Task Manager by either clicking on **Run** in the Windows XP Start Menu and typing **taskmgr.exe** and pressing ENTER, or in Windows Vista, just typing **taskmgr.exe** in the **Start Search** box in the Start Menu.

Click on the **Processes** tab, and then click on the **CPU** column twice so that the tasks are sorted by CPU usage. Near the top, you should see `ParticlesWin.exe`. On my machine, the unoptimized version is consuming around 12% to 14% of one core of the CPU. The version with the Xbox 360 optimizations consumes 8% to 10%. Clearly a pretty big win! And that was only removing the operator overloads from *one line* of the math code in the `Update` function.

There are other operator overloads in the loop that can be removed to improve performance even further. Why? Well, more performance means more particles, and more particles just looks more cool. See how many particles you can get running before you start to hurt the frame rate on Windows. On Xbox 360, you can easily get 20,000 particles running at 60 fps at 800x600 resolution. Somewhere between 20,000 and 30,000 particles, you hit a known issue that will crash the Xbox 360 back to the launcher. We are overrunning a limit that will be fixed in our next release. But 20,000 particles on the Xbox 360 at 60 Hz is a far cry from where started just a few pages ago. And the optimizations were easy to do!

### **Epilogue**

So, this white paper solves all performance issues on Windows and Xbox 360. Not. The purpose of the paper wasn't to do that. What I really wanted to do was give you the "tools" you need that are at your disposal today so that you can fix the problems as they come up. This is a recurring theme with XNA Game Studio development. I got it working on Windows, works fine; ported it to Xbox 360, sucks. This

white paper should help you identify the problems and optimize the code to make those problems go away on Xbox 360 and even improve your Windows performance at the same time.