ParticlesGPU Library Documentation

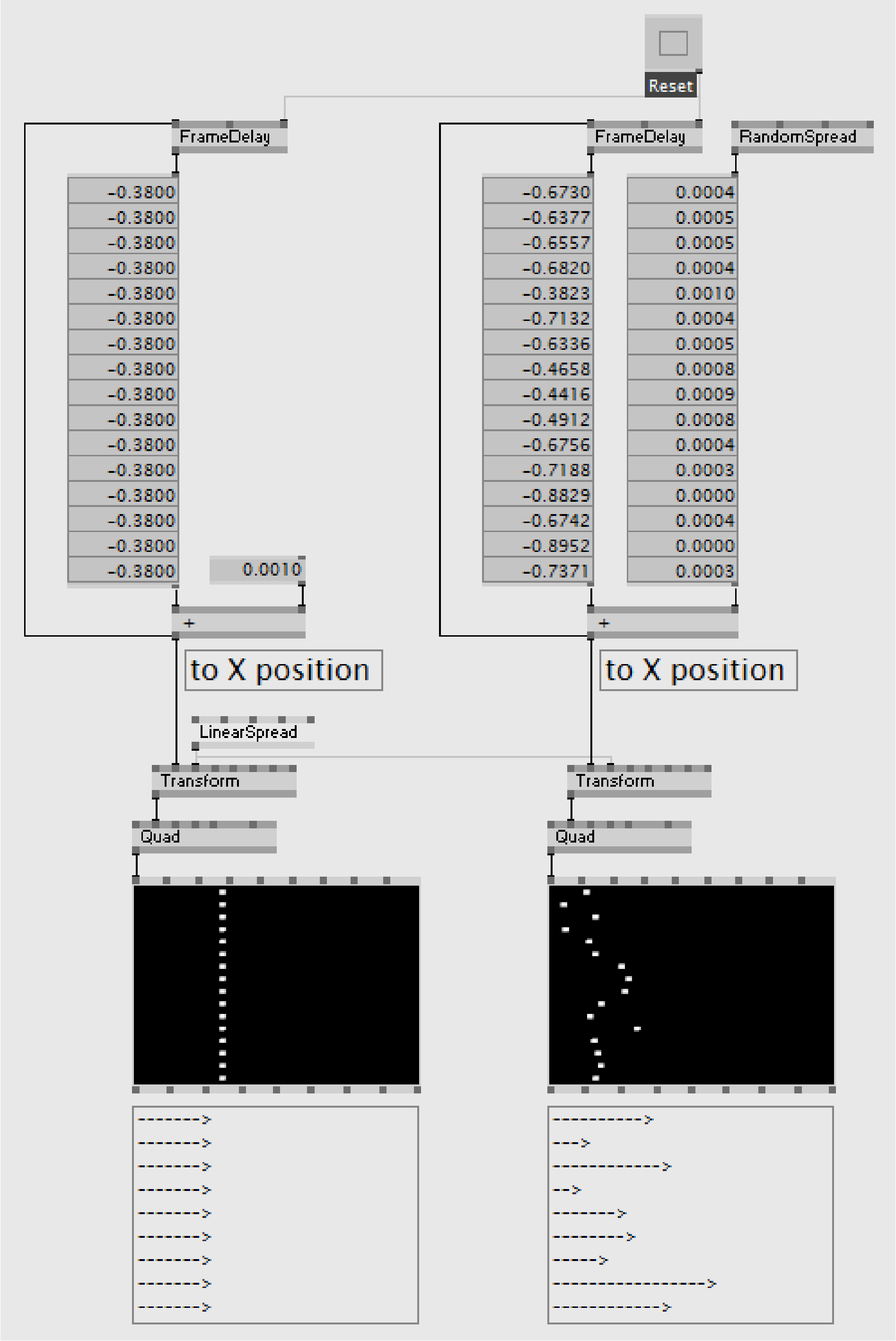
Natan Sinigaglia | dottore

Finally I found the time to think and write a proper document about my ParticlesGPU library. In the first part I'll try to explain the main principles of this approach to particles in GPU and after I'll go deeper in details on specific featured behaviours.

Credits to:
-Michael Mehling, who teached me many hlsl indispensable functions (holy tex2Dlod) while we were at Node08.
-Viktor Vicsek, who let me discover GPU Sprites function.
-Tonfilm, for many usefull hlsl transform functions taken from his ShaderTransform and for his Bicubic resample shader.

Contents:

Particle System using textures



- A particle system is a group of single objects (particles) that actually do something. We could build a simple particle system in vvvv by creating a spread of values (particle's position) that increase by a step value added every frame.

- As you see in this patch we can add the same value to all the particle's position (from framedelay) and they will move all at the same velocity. If we add different values to each particle, they will move differently, depending on how big is the step value for each one.

- What is important to understand is that in both these cases, the particle system works in a parallel way; it just apply the same function (+) to a list of data. (The difference of the second system is just in the values we put into the function +, the structure itself remains the same)

- So, a particle system is a parallel structured system.

  If we would like to have 500.000 particles we should have a spread of 500.000 slices and apply the function + to each slice, witch is quite heavy to handle in cpu/patch. High spread count = low performance.

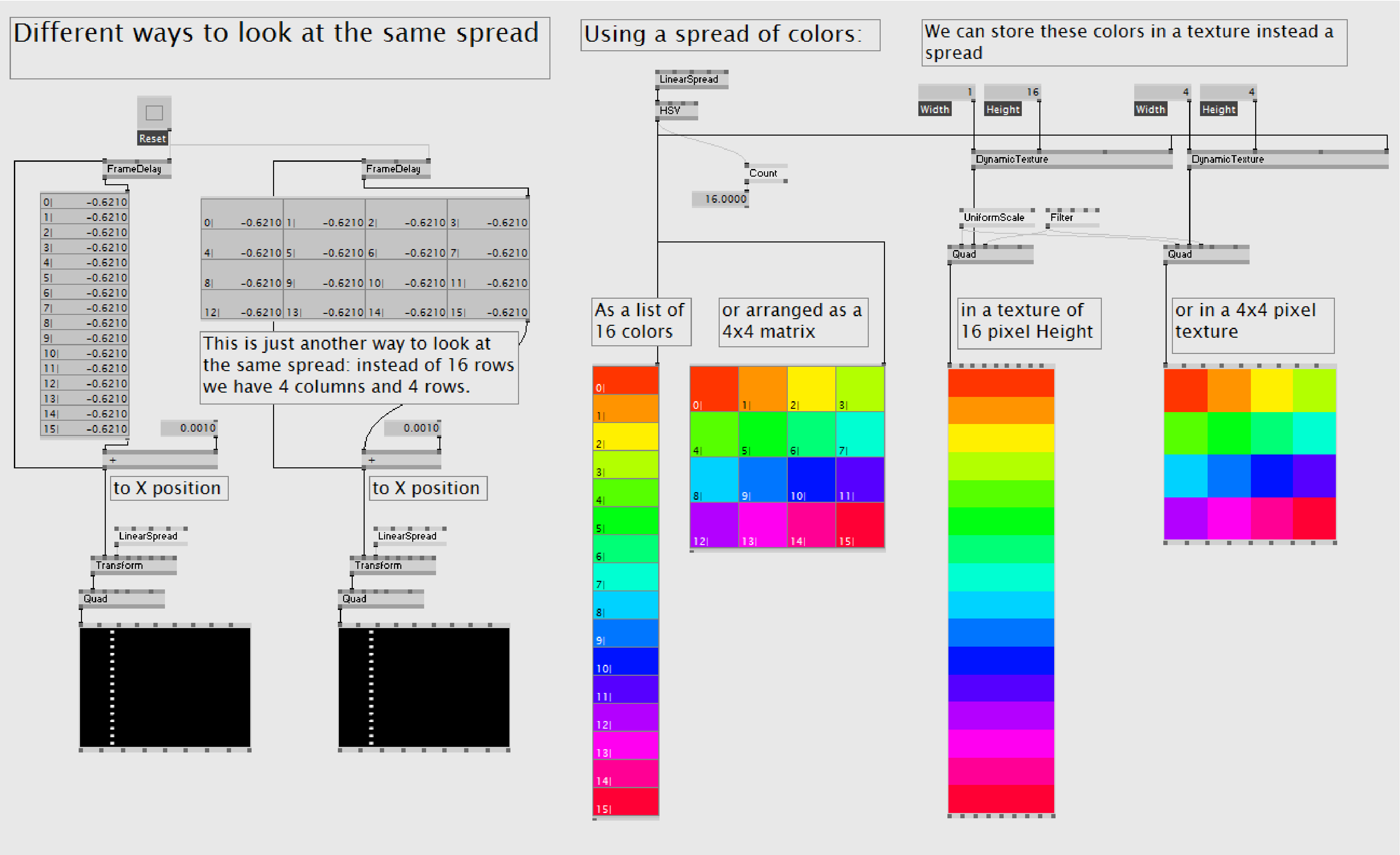
- But what is a spread? Just a container of data.

  Thinking at others way to manage big data containers i realized a stupid simple thing:

  What is a texture? Just a container of data.

  And then i thought: GPU is a piece of hardware that easily manages high resolution textures! It can work with a lot of data (pixel) in a parallel way! Just what we need for a particle system.

  In the following page i'll show you the similitude between spreads and textures as data containers.



Different ways to look at the same spread

0	-0.6210
1	-0.6210
2	-0.6210
3	-0.6210
4	-0.6210
5	-0.6210
6	-0.6210
7	-0.6210
8	-0.6210
9	-0.6210
10	-0.6210
11	-0.6210
12	-0.6210
13	-0.6210
14	-0.6210
15	-0.6210

0 -0.6210	1 -0.6210	2 -0.6210	3 -0.6210
4 -0.6210	5 -0.6210	6 -0.6210	7 -0.6210
8 -0.6210	9 -0.6210	10 -0.6210	11 -0.6210
12 -0.6210	13 -0.6210	14 -0.6210	15 -0.6210

This is just another way to look at the same spread: instead of 16 rows we have 4 columns and 4 rows.

to X position

to X position

Using a spread of colors:

As a list of 16 colors

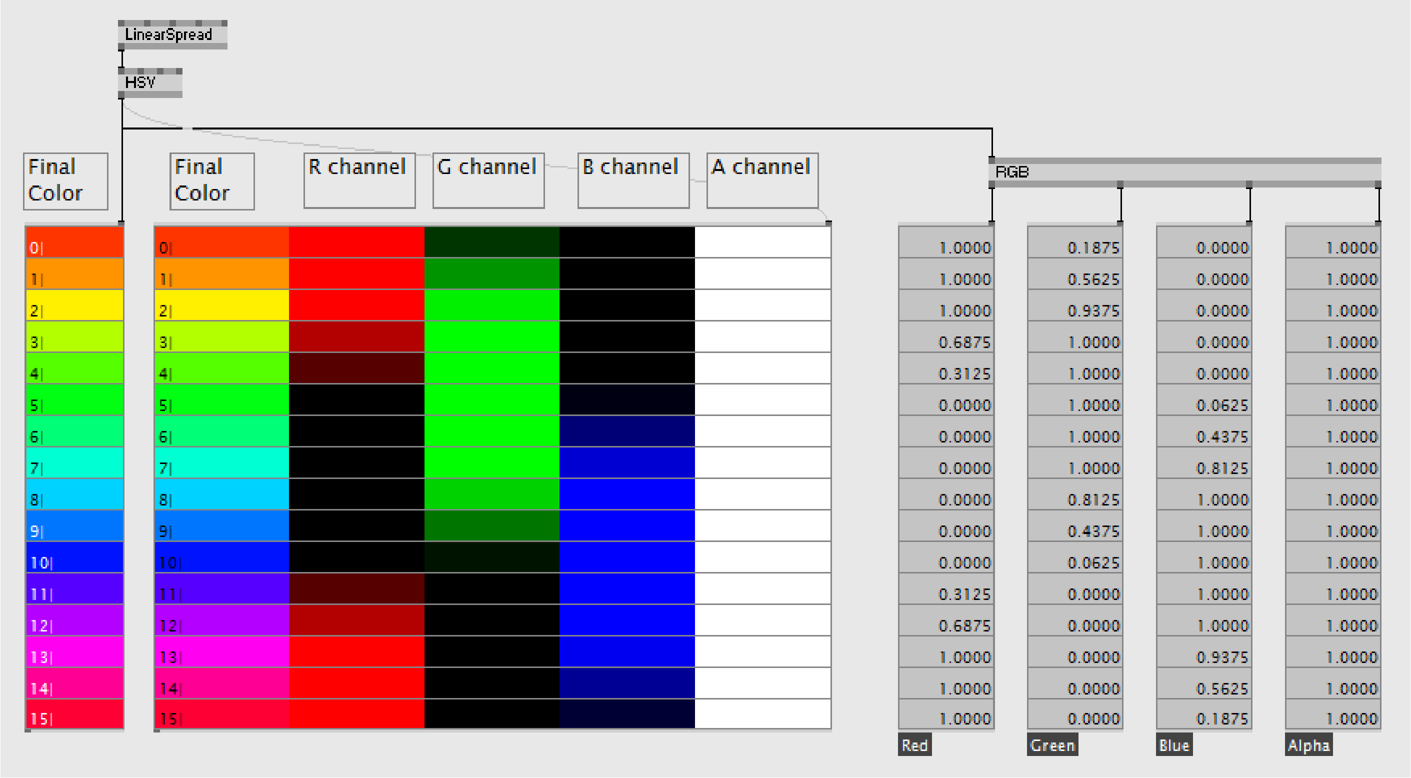
or arranged as a 4x4 matrix

We can store these colors in a texture instead a spread

in a texture of 16 pixel Height

or in a 4x4 pixel texture

Here is clear that we can use texture in the same way as spread, filling them with data



Red	Green	Blue	Alpha
1.0000	0.1875	0.0000	1.0000
1.0000	0.5625	0.0000	1.0000
1.0000	0.9375	0.0000	1.0000
0.6875	1.0000	0.0000	1.0000
0.3125	1.0000	0.0000	1.0000
0.0000	1.0000	0.0625	1.0000
0.0000	1.0000	0.4375	1.0000
0.0000	1.0000	0.8125	1.0000
0.0000	0.8125	1.0000	1.0000
0.0000	0.4375	1.0000	1.0000
0.0000	0.0625	1.0000	1.0000
0.3125	0.0000	1.0000	1.0000
0.6875	0.0000	1.0000	1.0000
1.0000	0.0000	0.9375	1.0000
1.0000	0.0000	0.5625	1.0000
1.0000	0.0000	0.1875	1.0000

A common image texture contains the informations to describe color for each pixel; every pixel contains 4 channels (4 numbers)

- -an 8bit number for Red
- -an 8bit number for Green
- -an 8bit number for Blue
- -an 8bit number for alpha

8bit number = 128 possible numbers (taken from 0 to 1, working in rgb space).

In a particle system we want to allow particles to move where they want, without limitations.

Let's try to use 8bit textures to store position of our particles. in a 2d particle system we would organize data as follow:

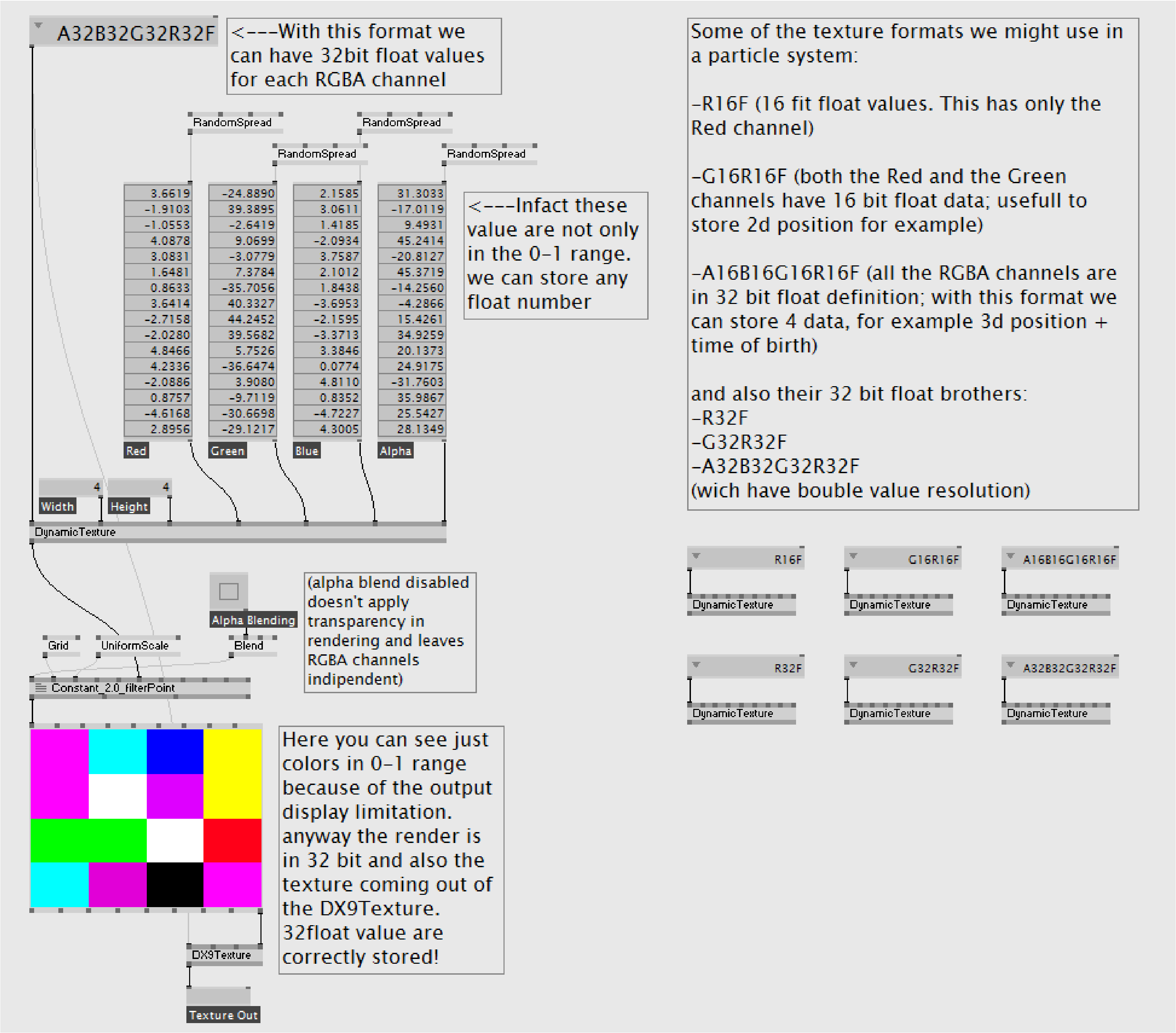
- Red = x position
- Green = y position

(we use just 2 channel of the pixels)

In this way, both x and y position will be values taken from those 128 number (in 0-1 space).This clearly is not enough for us; we need moooore then just 128 values for x and y! we want to be abled to push particles far away, for example to position x=13964,657 and y=-165,8465!

8bit describes a too small and too less definited space.

Fortunally, starting from shader model 2.0, directX supports float values rendering with 16bit and 32bit depth. Now we have high detailed values in texture and we can use it to store any kind of data!



<---With this format we can have 32bit float values for each RGBA channel

Red	Green	Blue	Alpha
3.6619	-24.8890	2.1585	31.3033
-1.9103	39.3895	3.0611	-17.0119
-1.0553	-2.6419	1.4185	9.4931
4.0878	9.0699	-2.0934	45.2414
3.0831	-3.0779	3.7587	-20.8127
1.6481	7.3784	2.1012	45.3719
0.8633	-35.7056	1.8438	-14.2560
3.6414	40.3327	-3.6953	-4.2866
-2.7158	44.2452	-2.1595	15.4261
-2.0280	39.5682	-3.3713	34.9259
4.8466	5.7526	3.3846	20.1373
4.2336	-36.6474	0.0774	24.9175
-2.0886	3.9080	4.8110	-31.7603
0.8757	-9.7119	0.8352	35.9867
-4.6168	-30.6698	-4.7227	25.5427
2.8956	-29.1217	4.3005	28.1349

<---Infact these value are not only in the 0-1 range. we can store any float number

(alpha blend disabled doesn't apply transparency in rendering and leaves RGBA channels indipendent)

Here you can see just colors in 0-1 range because of the output display limitation. anyway the render is in 32 bit and also the texture coming out of the DX9Texture. 32float value are correctly stored!

Some of the texture formats we might use in a particle system:

-R16F (16 fit float values. This has only the Red channel)

-G16R16F (both the Red and the Green channels have 16 bit float data; usefull to store 2d position for example)

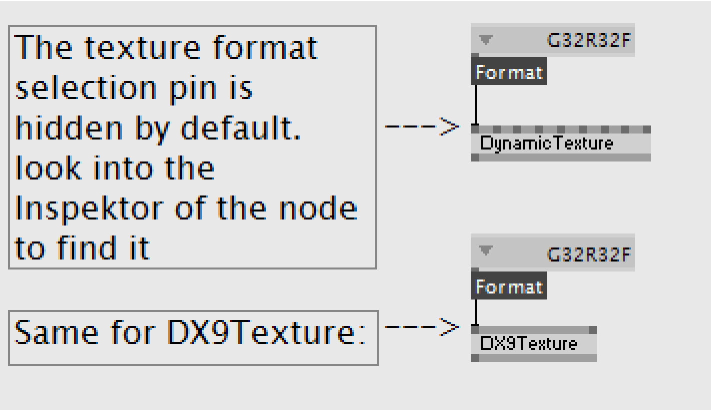
-A16B16G16R16F (all the RGBA channels are in 32 bit float definition; with this format we can store 4 data, for example 3d position + time of birth)

and also their 32 bit float brothers:
-R32F
-G32R32F
-A32B32G32R32F
(wich have bouble value resolution)

To allow any shader to be rendered in 16 or 32 bit is necessary to compile the pixel shader with Shader Model >= 2.0
Here the technique declaration inside the shader, where you choose shader model (it's at the end of hlsl code):

```
technique TConstant

{

   pass P0

   {

    VertexShader = compile vs_1_1 VS();

    PixelShader = compile ps_2_0 PS();

   }

}
```

remember to select the correct Texture format in all the Dynamic Texture nodes and also in all the DX9Texture, otherwise the default format will be in 8 bit per channel

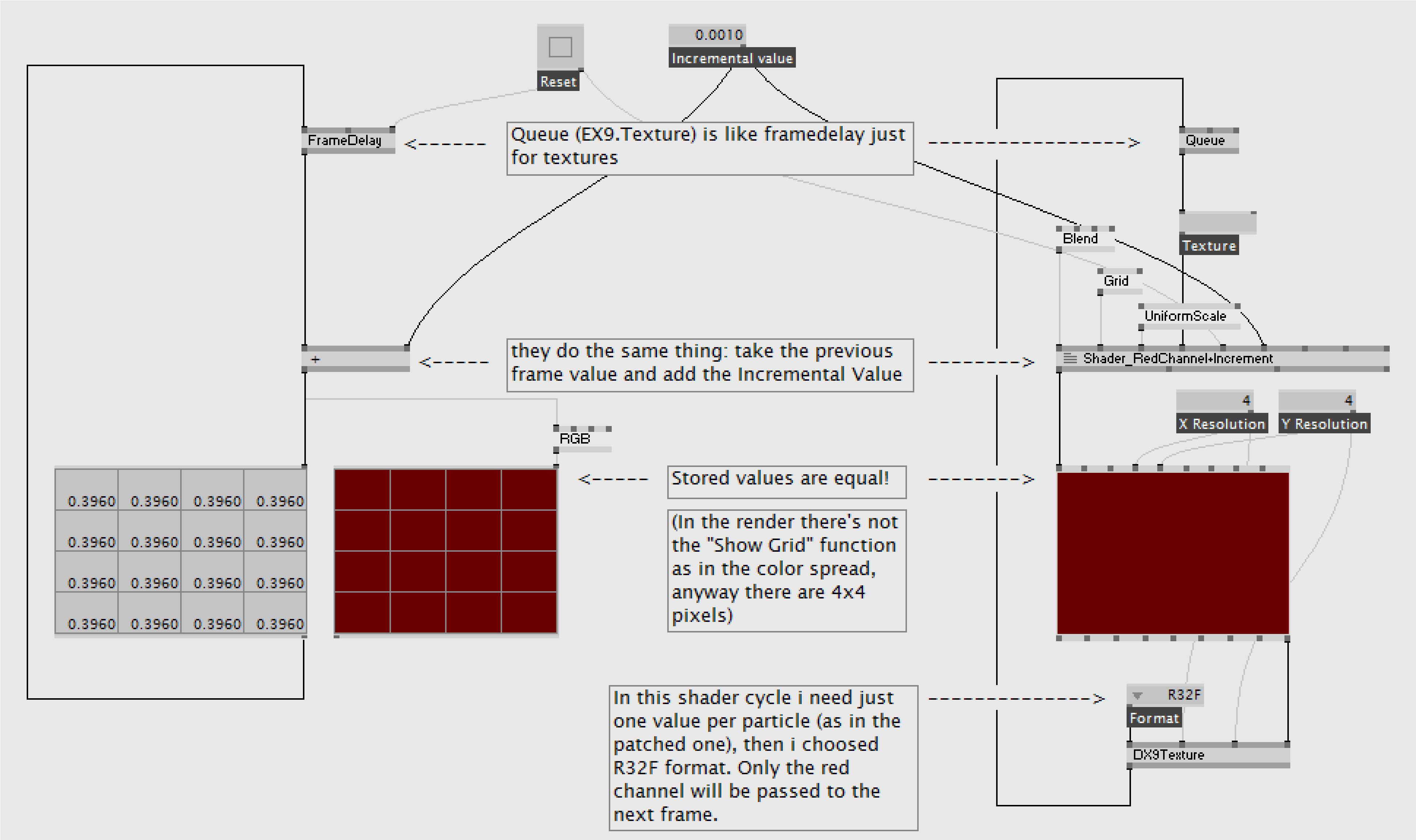


The texture format selection pin is hidden by default. look into the Inspektor of the node to find it

Same for DX9Texture:

Now that we have all the ingredients, let's try to build a basic Particle System using textures!

I'll do the same particle position cycle both in patch and in shader to show you that's really the same:



Queue (EX9.Texture) is like framedelay just for textures

they do the same thing: take the previous frame value and add the Incremental Value

0.3960	0.3960	0.3960	0.3960
0.3960	0.3960	0.3960	0.3960
0.3960	0.3960	0.3960	0.3960
0.3960	0.3960	0.3960	0.3960

Stored values are equal!

(In the render there's not the "Show Grid" function as in the color spread, anyway there are 4x4 pixels)

In this shader cycle i need just one value per particle (as in the patched one), then i choosed R32F format. Only the red channel will be passed to the next frame.

- We use a grid 2x2 resolution as input mesh in the shader: we just want a quad all over the screen
- UniformScale set to 2. In this way the grid will cover the entire render space from -1 to 1.
- Blend (EX9.RenderState Advanced): Alpha Blending pin set to 0. we don't want to evaluate trasparency in rendering.

Here is the hlsl code of the shader in the previous page

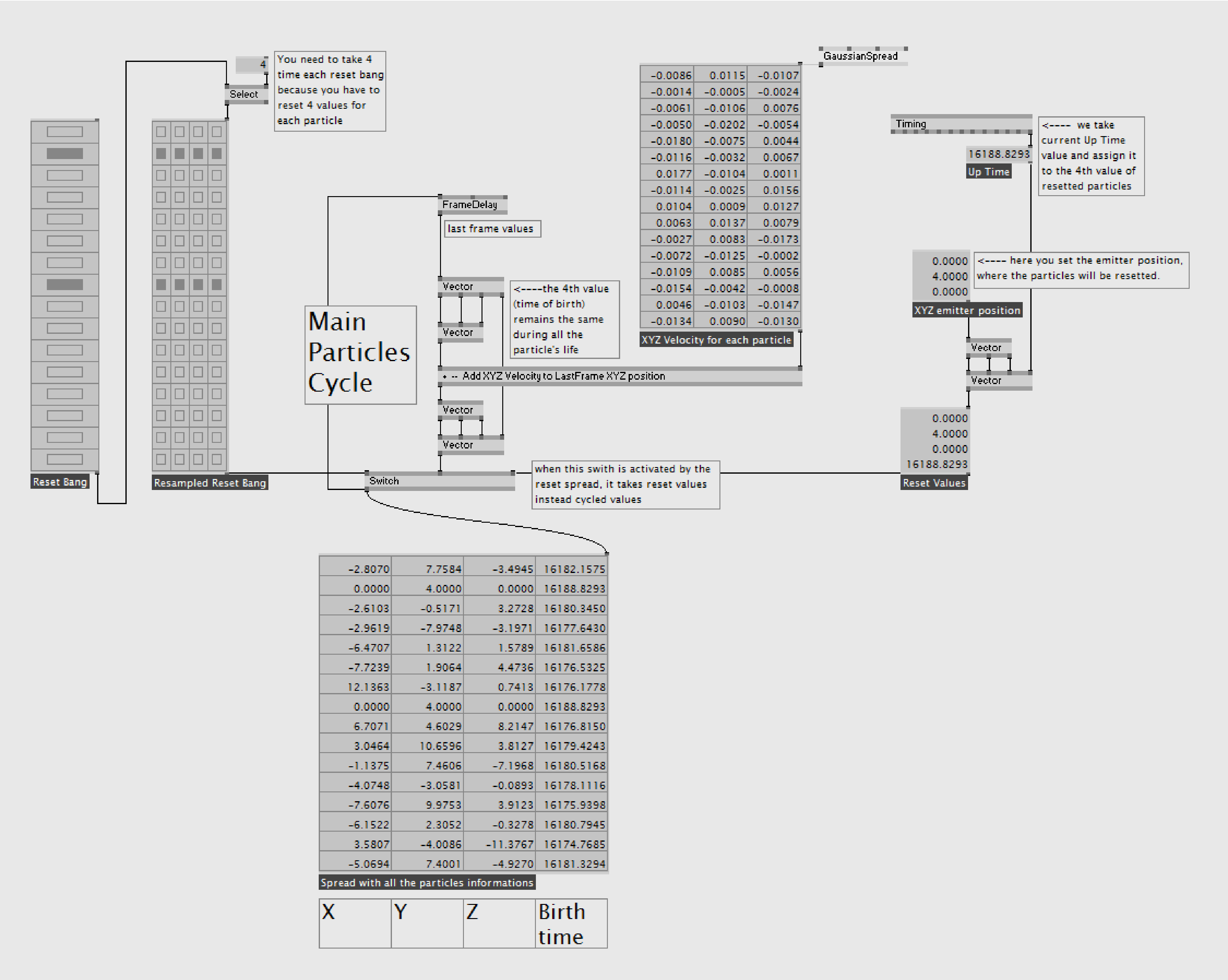
```
////// PARAMETERS:

//transforms
float4x4 tWVP: WORLDVIEWPROJECTION ;
//texture
texture Tex <string uiname="Texture";>;
sampler Samp = sampler_state //sampler for doing the texture-lookup
{
    Texture   = (Tex);       //apply a texture to the sampler
    MipFilter = none;        //sampler states
    MinFilter = none;
    MagFilter = none;
};
bool Reset;
float Increment; //value to add every frame; passed from the patch

struct vs2ps
{
    float4 Pos : POSITION ;
    float4 TexCd : TEXCOORD0 ;
};


////// VERTEXSHADERS


vs2ps VS(
    float4 Pos : POSITION ,
    float4 TexCd : TEXCOORD0 )
{
     //inititalize all fields of output struct with 0
    vs2ps Out = (vs2ps)0;
    //transform position
    Out.Pos = mul(Pos, tWVP);
    Out.TexCd = TexCd;
    return Out;
}
```

```
////// PIXELSHADERS:

float4 PS(vs2ps In): COLOR
{
    //take the red from the last frame texture and add "Increment"
    float newRed = tex2D(Samp, In.TexCd).r + Increment;
    // when reset the cycle:
    if(Reset) newRed = 0;
    return float4(newRed,0,0,0);
}


////// TECHNIQUES:

technique CycleRed
{
    pass P0
       {
        VertexShader = compile vs_1_1 VS();
        PixelShader = compile ps_2_0 PS();
       }
}
```

Let's do a more sofisticated particle system. This is what we want:

- *3 float values for 3d dimensional position (XYZ)*
- *Particle's birth time* (to obtain particle's life time, usefull for animations like alpha fade in and others...)
- *PP velocity* (different xyz velocity for each particle; these values will be added to the previous frame position, frame by frame)
- *PP Reset* (Per Particle reset; we want to be able to reset each particle individually)
- *Emitter xyz Position* (where the particles will be emitted on reset bang)

See in the next page how to do this in patch



You need to take 4 time each reset bang because you have to reset 4 values for each particle

Main Particles Cycle

last frame values

<----the 4th value (time of birth) remains the same during all the particle's life

-0.0086	0.0115	-0.0107
-0.0014	-0.0005	-0.0024
-0.0061	-0.0106	0.0076
-0.0050	-0.0202	-0.0054
-0.0180	-0.0075	0.0044
-0.0116	-0.0032	0.0067
0.0177	-0.0104	0.0011
-0.0114	-0.0025	0.0156
0.0104	0.0009	0.0127
0.0063	0.0137	0.0079
-0.0027	0.0083	-0.0173
-0.0072	-0.0125	-0.0002
-0.0109	0.0085	0.0056
-0.0154	-0.0042	-0.0008
0.0046	-0.0103	-0.0147
-0.0134	0.0090	-0.0130

XYZ Velocity for each particle

<---- we take current Up Time value and assign it to the 4th value of resetted particles

<---- here you set the emitter position, where the particles will be resetted.

when this swith is activated by the reset spread, it takes reset values instead cycled values

X	Y	Z	Birth time
-2.8070	7.7584	-3.4945	16182.1575
0.0000	4.0000	0.0000	16188.8293
-2.6103	-0.5171	3.2728	16180.3450
-2.9619	-7.9748	-3.1971	16177.6430
-6.4707	1.3122	1.5789	16181.6586
-7.7239	1.9064	4.4736	16176.5325
12.1363	-3.1187	0.7413	16176.1778
0.0000	4.0000	0.0000	16188.8293
6.7071	4.6029	8.2147	16176.8150
3.0464	10.6596	3.8127	16179.4243
-1.1375	7.4606	-7.1968	16180.5168
-4.0748	-3.0581	-0.0893	16178.1116
-7.6076	9.9753	3.9123	16175.9398
-6.1522	2.3052	-0.3278	16180.7945
3.5807	-4.0086	-11.3767	16174.7685
-5.0694	7.4001	-4.9270	16181.3294

Spread with all the particles informations

How to do the same in Shader?

- XYZ position values for each particle:

  We use RGB channels of a 32bit float texture to store X,Y,Z in pixels. Every pixel is a particle -->

- Birth Time for each pixel-particle:

  We give the UpTime value to the shader, so we'll assign it to the A channel of resetted pixels/particles inside the pixel shader.

- XYZ velocity for every pixel-particle:

  We feed our ParticlesCycle shader (that will manage the texture cycle) with a velocity texture: we need 3 values for xyz velocity and we need quite high value resolution to use also small velocity values; for this reason we'll use an RGBA 16bit float texture format (there isn't a float texture format with 3 channels, simply we won't use the A channel).

- PP reset:

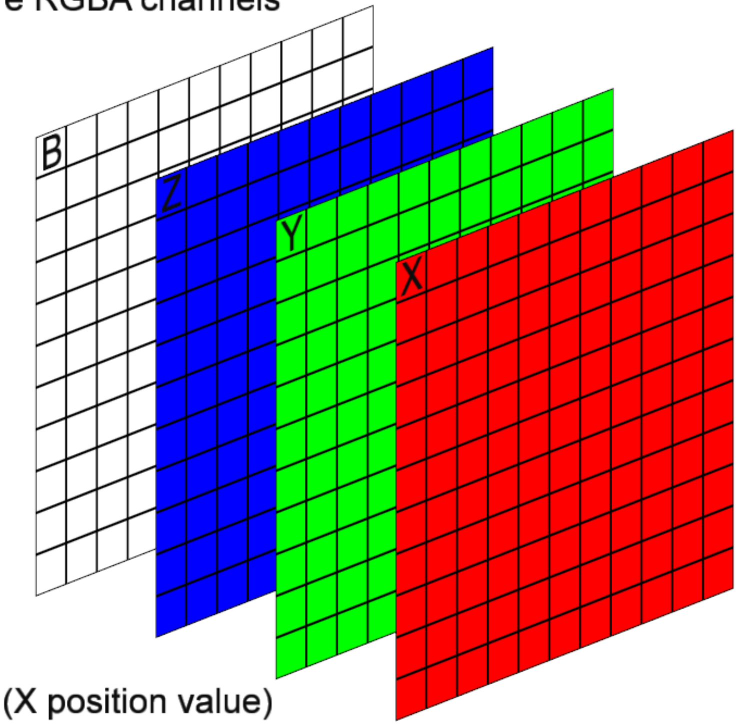
  We feed the ParticlesCycle shader also with a bang texture: each pixel of this texture will tell to the relative pixel of the cycle when is resetted. For this texture we just need a boolean value (0/1) so we can just use a simple format like 8A (just the alpha channel in 8 bit depth).

  Note: we could write the PPreset bang values in the A channel of the velocity texture, optimizing and saving texture lookups in PixelShader. I decided to keep separated for didactic reasons.

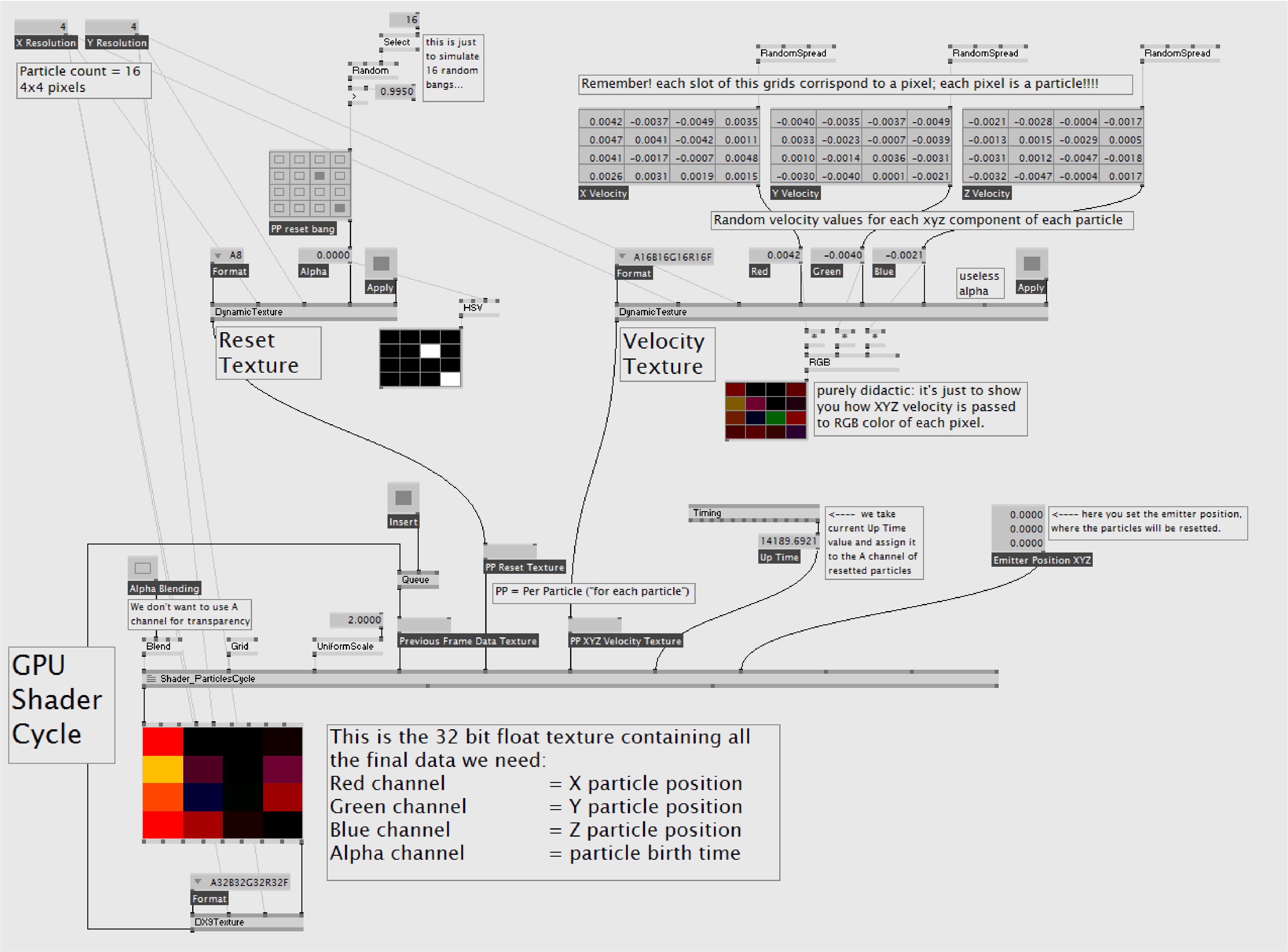
- Control emission XYZ position:

  We provide xyz emitter position to the shader in order to place resetted particles where we want (we'll assign the resetted position inside the pixel shader)

Texture RGBA channels



R = X (X position value)
G = Y (Y position value)
B = Z (Z position value)
A = B (Particle's birth time)



Particle count = 16
4x4 pixels

this is just to simulate 16 random bangs...

Remember! each slot of this grids corrispond to a pixel; each pixel is a particle!!!!

X Velocity

0.0042	-0.0037	-0.0049	0.0035
0.0047	0.0041	-0.0042	0.0011
0.0041	-0.0017	-0.0007	0.0048
0.0026	0.0031	0.0019	0.0015

Y Velocity

-0.0040	-0.0035	-0.0037	-0.0049
0.0033	-0.0023	-0.0007	-0.0039
0.0010	-0.0014	0.0036	-0.0031
-0.0030	-0.0040	0.0001	-0.0021

Z Velocity

-0.0021	-0.0028	-0.0004	-0.0017
-0.0013	0.0015	-0.0029	0.0005
-0.0031	0.0012	-0.0047	-0.0018
-0.0032	-0.0047	-0.0004	0.0017

Random velocity values for each xyz component of each particle

Reset Texture

Velocity Texture

useless alpha

purely didactic: it's just to show you how XYZ velocity is passed to RGB color of each pixel.

<---- we take current Up Time value and assign it to the A channel of resetted particles

<---- here you set the emitter position, where the particles will be resetted.

PP = Per Particle ("for each particle")

We don't want to use A channel for transparency

GPU Shader Cycle

This is the 32 bit float texture containing all the final data we need:

Red channel = X particle position
Green channel = Y particle position
Blue channel = Z particle position
Alpha channel = particle birth time

Here is the hlsl code of the ParticlesCycle Shader in the previous page

```
// PARAMETERS:
//transforms
float4x4 tWVP: WORLDVIEWPROJECTION ;

texture TexPrev <string uiname="Previous Frame Data Texture";>;
sampler SampPrev = sampler_state//sampler for doing the texture-lookup
{
    Texture   = (TexPrev);       //apply a texture to the sampler
    MipFilter = none;            //sampler states
    MinFilter = none;
    MagFilter = none;
};


texture TexReset <string uiname="PP Reset Texture";>;
sampler SampReset = sampler_state
{
    Texture   = (TexReset);
    MipFilter = none;
    MinFilter = none;
    MagFilter = none;
};


texture TexVel <string uiname="PP XYZ Velocity Texture";>;
sampler SampVel = sampler_state
{
    Texture   = (TexVel);
    MipFilter = none;
    MinFilter = none;
    MagFilter = none;
};
float UpTime;
float3 ResetPos <string uiname="Emitter Position";>;

struct vs2ps
{
    float4 Pos : POSITION ;
    float4 TexCd : TEXCOORD0 ;
};
```

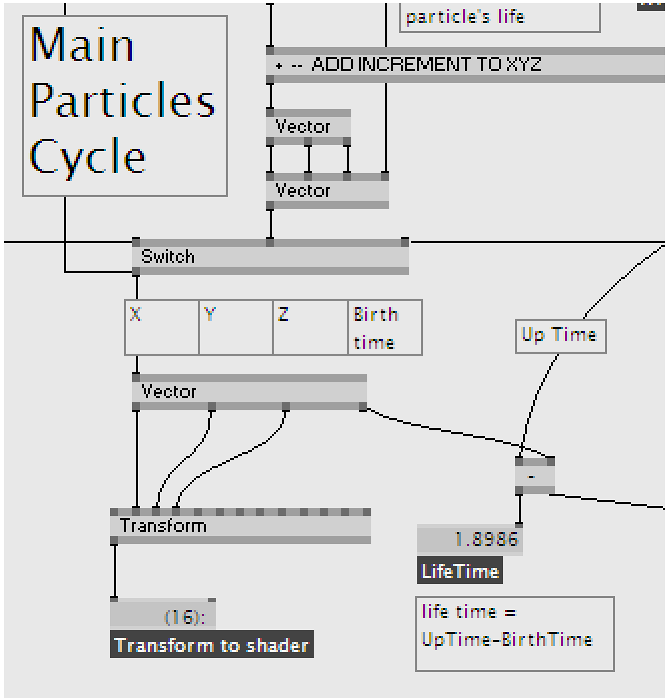
```
// VERTEXSHADERS
vs2ps VS(
    float4 Pos : POSITION ,
    float4 TexCd : TEXCOORD0 )
{
    vs2ps Out = (vs2ps)0;
    Out.Pos = mul(Pos, tWVP);  //transform position
    Out.TexCd = TexCd;
    return Out;
}
// PIXELSHADERS:
float4 PS(vs2ps In): COLOR
{
    // take the RGBA values from the last frame texture
    float4 lastFrame = tex2D(SampPrev, In.TexCd);
    // get the reset bang info from the alpha channel of the Reset Texture
    bool reset = tex2D(SampReset, In.TexCd).a > 0.5;
    // get the XYZ velocity values from the RGB channels of Velocity Texture
    float3 vel = tex2D(SampVel, In.TexCd).rgb;
    // new RGBA data
    float4 newData = 0;
    /////// GPU CYCLE:
    if(reset)  //set RGB to the reset position and A to the birth time
      { newData = float4(ResetPos, UpTime); }
    else  //get the old xyz and add xyz velocity; pass the birthTime to A
      { newData = float4(lastFrame.rgb + vel, lastFrame.a); }
    return newData;
}
// TECHNIQUES:
technique RGBA_Cycle
{
    pass P0
      {
        VertexShader = compile vs_1_1 VS();
        PixelShader = compile ps_2_0 PS();
      }
}
```

Reading Data Texture inside a Shader

Well, just 5 min ago my laptop's graphic card said goodbye while I was writing in OpenOffice (be carefully, It really push the hardware over the limit). This mean from now on there will be less (none) render screenshots and more wonderful drawings.

In the last 13 pages we saw how to use texture as dynamic containers for any kind of data. Now we'll focus on how to use these texture.

In CPU particle system (pg.10) we would use particles data (stored in the spread) as following:



- take the spread containing all the data (XYZB)
- divide it (With a vector4D split)
- Use XYZ in a Transform node and provide it to any shader
- Obtain the LifeTime (Current UpTime – BirthTime)

really easy so far....

Let's discuss in the next page how to retrieve data from a texture and use it in our ParticlesGPU shaders

We want to use Data Texture for:

- Transform geometry (translate, scale,..) (like we usually do with transforms)
- Control shading parameters (example: using the birth time to influence the color)

To achieve this inside a shader we work respectively:

- In Vertex Shader to transform the geometry (VS works on vertices)
- In Pixel Shader to adjust the shading of geometry (PS works on pixels that cover the rendered geometry = what you actually see)

Both in VS and PS we need to read data from the Data Texture of the shader cycle:

- First of all we need to declare the texture in the declaration part of the code (the beginning); we also create a sampler that samples the texture (VS and PS will call this sampler to read the texture).

```
texture TexData <string uiname="Data Texture";>;
sampler SampData = sampler_state  //sampler for doing the texture-lookup
{
    Texture   = (TexData);
    MipFilter = none;
    MinFilter = none;
    MagFilter = none;
};
```

how to access DataTexture RGBA information inside VS and PS:

- In **Vertex Shader** we use the function `tex2Dlod(Sampler,TexCoord)`

  As we are in the vertex shader, we work with vertices; each vertex, using the `tex2Dlod` function, will have access to the Data Texture and will be able to use those values to transform itself. Example:

  ```
  Pos.xyz += tex2Dlod(Samp,TexCd).rgb;
  ```

  this line of code simply tells: “take the current vertex XYZ position (in the object space) and add to it the values stored in RGB channels of one particular point of this texture (the point is indicated by vertex's TexCd)”

  VS applies this operation to all the vertices coming from the mesh.

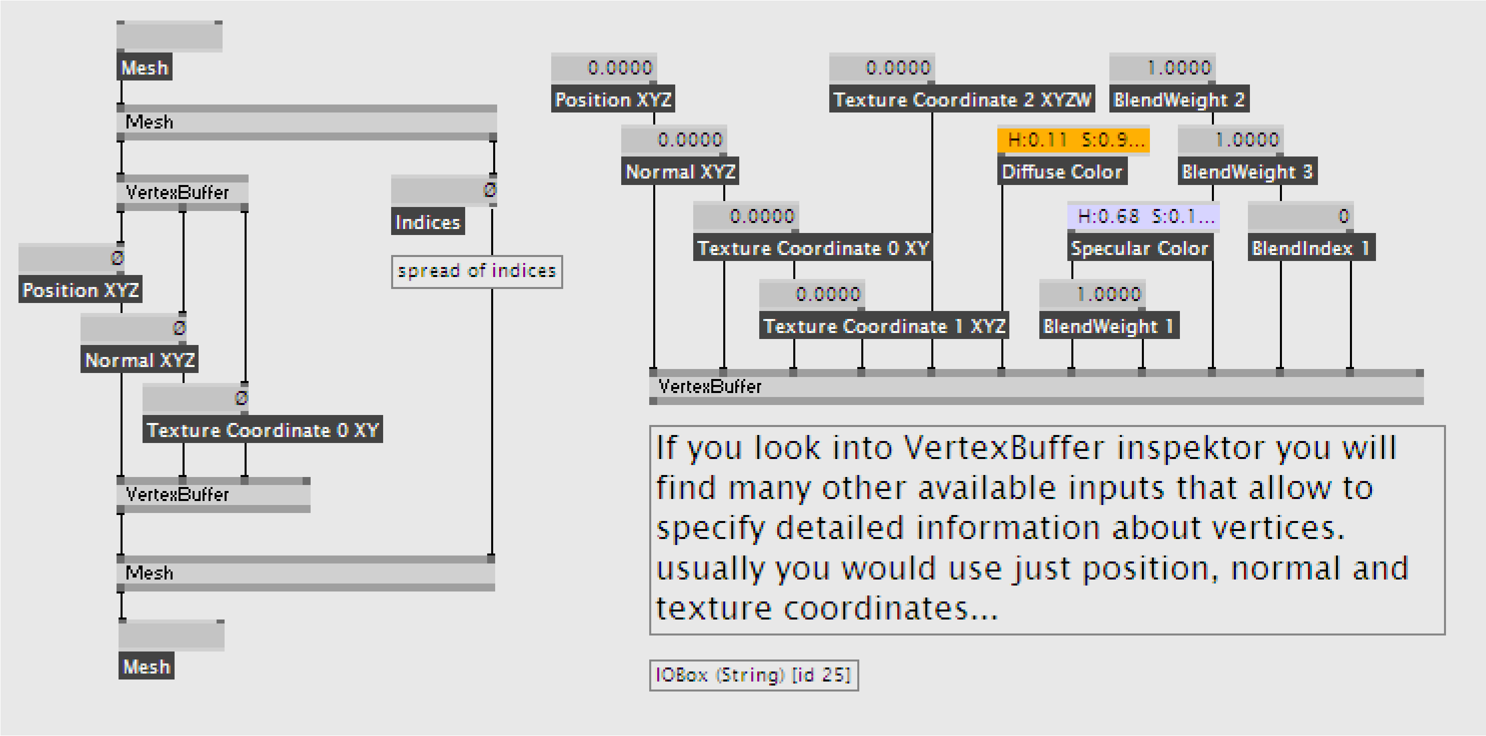
- In **Pixel Shader** we use the common `tex2D(Sampler,TexCoord)` function. I said common because this is the function always used to apply a texture onto a geometry.

In the next pages I'll explain how to prepare meshes in a clever way to be controlled by values in the Data Texture.

Mega Mesh

What's inside a Mesh?

- <u>Vertex Buffer:</u> where all the information about each vertex of the mesh are stored. directx documentation for a detailed information about vertexbuffers
- <u>Indices</u>: in directx the basic element (polygon) that compounds surface is a triangle face. Indexbuffer defines which 3 vertices in the vertexbuffer make up each triangle face of the mesh.



If you look into VertexBuffer inspektor you will find many other available inputs that allow to specify detailed information about vertices. usually you would use just position, normal and texture coordinates...

As you probably noticed, in ParticlesGPU examples there are dedicated modules that generate a single huge mesh for all the particles. Let's explain why:

In a common **CPU approach** (like we saw at pg.14) we use particles data (spreads) to feed a Transform node.
When a shader receive a spread (can be spread of transforms, colors, textures, ...any other input pin of the shader), this happens EVERY FRAME:

- It loads (“call”) the mesh from CPU memory (it's a slow operation).
- It applies VS to the mesh (using the first slice from inputs)
- It applies PS to those pixels interested by the surface (using the first slice from inputs)

Then it repeats these steps using the second slice value from input spread. And it goes this way throw all the slices of the spread.

This means if we have 5000 objects (a spread of 5000 in the Transform node), the shader will repeat the entire pipeline 5000 times each frame!
It will call 5000 times the mesh from CPU, every frame! This is a real bottleneck.
...that's why your pc start to cry and performances fall down using Transform with big spreads...
Clearly this approach is impracticable if we have in mind to render up to 1 million of objects. We need to reduce the bottleneck.

The solution is to build a single big mesh containing all the particles that will be rendered.

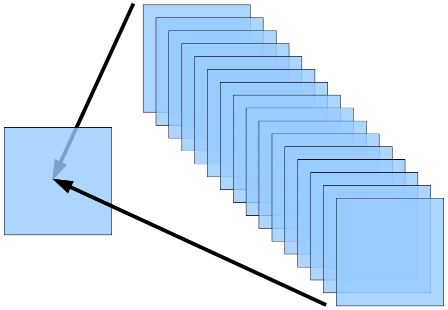
if we want 5000 quad particles, our Mega Mesh will contain 5000 overlapping quads written as a single geometry.
The shader will render 5000 particles, calling the mesh just one time per frame. There's a massive performance improvement...

In the next page I'll show you how to build this unique Mega Mesh.

Let's take as reference our particle system builded in pg.12
We want to build a mesh containing all the 16 particles evaluated in the Data Texture coming out of the Shader Cycle. We decide each particle is a quad.

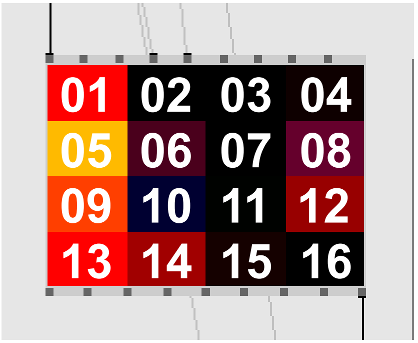
The Mega Mesh will contain:

- XYZ vertices position of all the 16 quads (there will be 16 quads in the same position {0,0}).
- Texture Coordinates 0 to retrieve informations (xyz position, birth time) from the Data Texture.
- Texture Coordinates 1 to apply an image texture onto the quads (these texCoord are obviously different from the previous one)
- Indices in order to build quads faces.



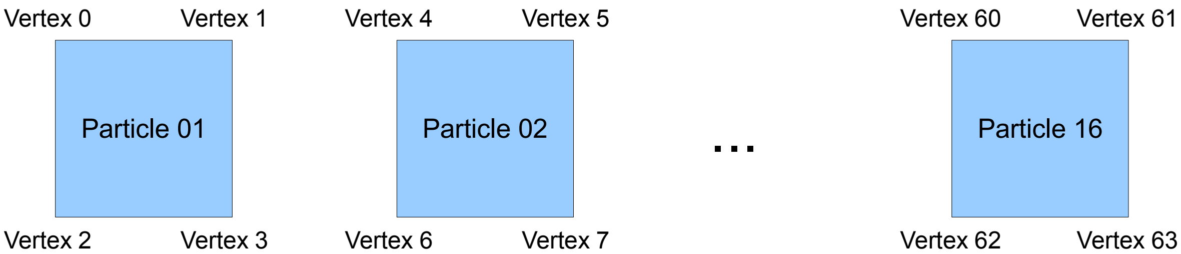
All the 16 Quads overlapping in the same position

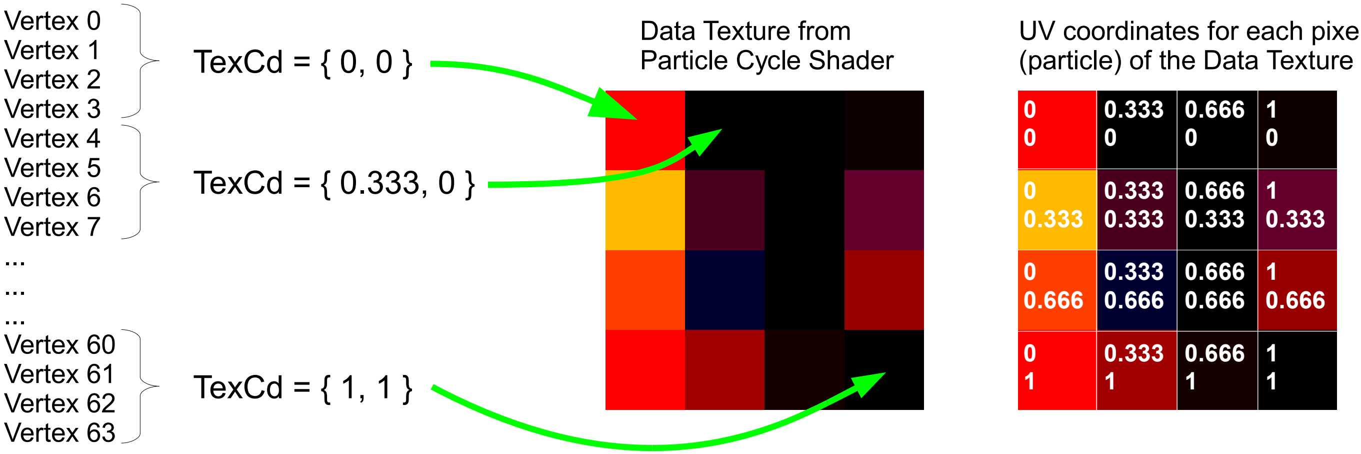
About Texture Coordinates 0:
All the 4 vertices of each quad need a TexCoord information that tells them wich pixel of the Data Texture they will sample. All the 4 vertices will sample the same pixel (otherwise they will move differently and the quad shape will deform) => they will have the same TexCoord.



01	02	03	04
05	06	07	08
09	10	11	12
13	14	15	16

Each Particle corresponds to a certain pixel in Data Texture





Vertex 0
Vertex 1
Vertex 2
Vertex 3
TexCd = { 0, 0 }

Vertex 4
Vertex 5
Vertex 6
Vertex 7
TexCd = { 0.333, 0 }

...
...
...

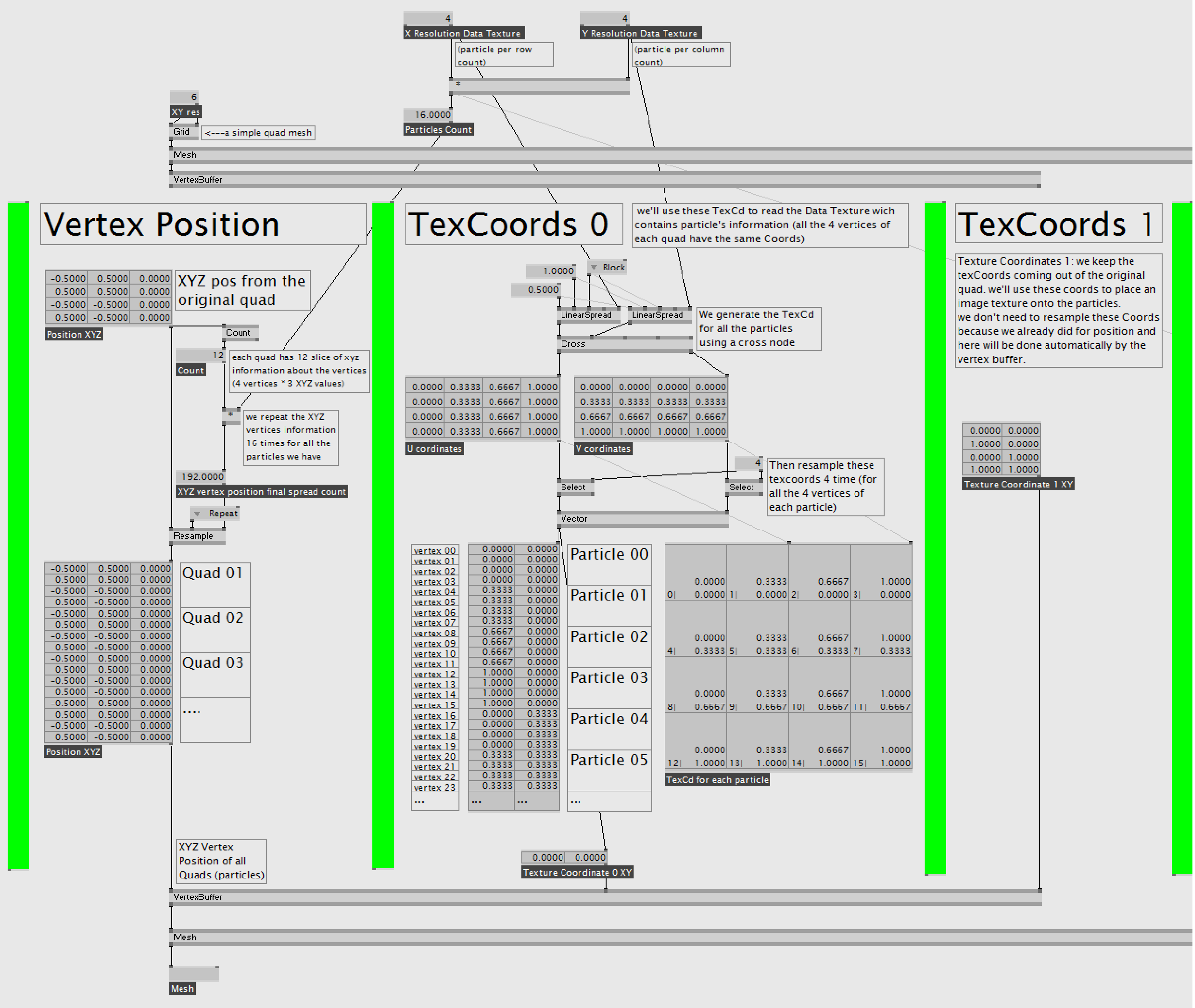
Vertex 60
Vertex 61
Vertex 62
Vertex 63
TexCd = { 1, 1 }

Data Texture from Particle Cycle Shader

UV coordinates for each pixel (particle) of the Data Texture

0 0	0.333 0	0.666 0	1 0
0 0.333	0.333 0.333	0.666 0.333	1 0.333
0 0.666	0.333 0.666	0.666 0.666	1 0.666
0 1	0.333 1	0.666 1	1 1

Look at next 2 pages to see the Mega Mesh Module



(particle per row count)

(particle per column count)

<---a simple quad mesh

Vertex Position

TexCoords 0

we'll use these TexCd to read the Data Texture wich contains particle's information (all the 4 vertices of each quad have the same Coords)

TexCoords 1

Texture Coordinates 1: we keep the texCoords coming out of the original quad. we'll use these coords to place an image texture onto the particles.
we don't need to resample these Coords because we already did for position and here will be done automatically by the vertex buffer.

XYZ pos from the original quad

-0.5000	0.5000	0.0000
0.5000	0.5000	0.0000
-0.5000	-0.5000	0.0000
0.5000	-0.5000	0.0000

Position XYZ

each quad has 12 slice of xyz information about the vertices (4 vertices * 3 XYZ values)

we repeat the XYZ vertices information 16 times for all the particles we have

-0.5000	0.5000	0.0000
0.5000	0.5000	0.0000
-0.5000	-0.5000	0.0000
0.5000	-0.5000	0.0000
-0.5000	0.5000	0.0000
0.5000	0.5000	0.0000
-0.5000	-0.5000	0.0000
0.5000	-0.5000	0.0000
-0.5000	0.5000	0.0000
0.5000	0.5000	0.0000
-0.5000	-0.5000	0.0000
0.5000	-0.5000	0.0000
-0.5000	0.5000	0.0000
0.5000	0.5000	0.0000
-0.5000	-0.5000	0.0000
0.5000	-0.5000	0.0000

Position XYZ

Quad 01

Quad 02

Quad 03

....

XYZ Vertex Position of all Quads (particles)

We generate the TexCd for all the particles using a cross node

0.0000	0.3333	0.6667	1.0000
0.0000	0.3333	0.6667	1.0000
0.0000	0.3333	0.6667	1.0000
0.0000	0.3333	0.6667	1.0000

U cordinates

0.0000	0.0000	0.0000	0.0000
0.3333	0.3333	0.3333	0.3333
0.6667	0.6667	0.6667	0.6667
1.0000	1.0000	1.0000	1.0000

V cordinates

Then resample these texcoords 4 time (for all the 4 vertices of each particle)

vertex 00	0.0000	0.0000	Particle 00
vertex 01	0.0000	0.0000	
vertex 02	0.0000	0.0000	
vertex 03	0.0000	0.0000	
vertex 04	0.3333	0.0000	Particle 01
vertex 05	0.3333	0.0000	
vertex 06	0.3333	0.0000	
vertex 07	0.3333	0.0000	
vertex 08	0.6667	0.0000	Particle 02
vertex 09	0.6667	0.0000	
vertex 10	0.6667	0.0000	
vertex 11	0.6667	0.0000	
vertex 12	1.0000	0.0000	Particle 03
vertex 13	1.0000	0.0000	
vertex 14	1.0000	0.0000	
vertex 15	1.0000	0.0000	
vertex 16	0.0000	0.3333	Particle 04
vertex 17	0.0000	0.3333	
vertex 18	0.0000	0.3333	
vertex 19	0.0000	0.3333	
vertex 20	0.3333	0.3333	Particle 05
vertex 21	0.3333	0.3333	
vertex 22	0.3333	0.3333	
vertex 23	0.3333	0.3333	
...	...	...	...

0| 0.0000 0.0000	1| 0.3333 0.0000	2| 0.6667 0.0000	3| 1.0000 0.0000
4| 0.0000 0.3333	5| 0.3333 0.3333	6| 0.6667 0.3333	7| 1.0000 0.3333
8| 0.0000 0.6667	9| 0.3333 0.6667	10| 0.6667 0.6667	11| 1.0000 0.6667
12| 0.0000 1.0000	13| 0.3333 1.0000	14| 0.6667 1.0000	15| 1.0000 1.0000

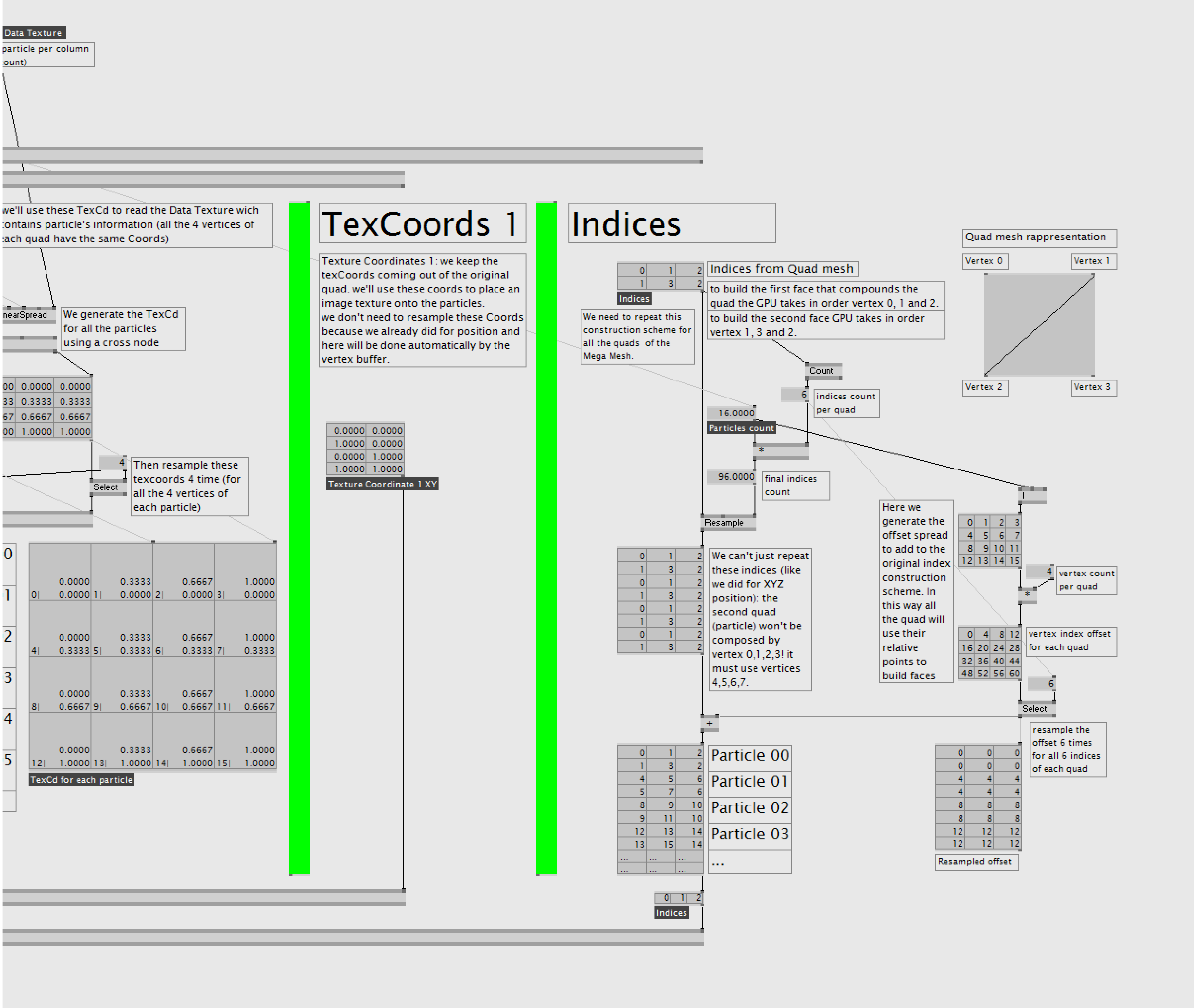
TexCd for each particle

0.0000	0.0000

Texture Coordinate 0 XY

0.0000	0.0000
1.0000	0.0000
0.0000	1.0000
1.0000	1.0000

Texture Coordinate 1 XY



particle per column
ount)

we'll use these TexCd to read the Data Texture wich ontains particle's information (all the 4 vertices of ach quad have the same Coords)

We generate the TexCd for all the particles using a cross node

00	0.0000	0.0000
33	0.3333	0.3333
67	0.6667	0.6667
00	1.0000	1.0000

Then resample these texcoords 4 time (for all the 4 vertices of each particle)

0| 0.0000 0.0000	1| 0.3333 0.0000	2| 0.6667 0.0000	3| 1.0000 0.0000
4| 0.0000 0.3333	5| 0.3333 0.3333	6| 0.6667 0.3333	7| 1.0000 0.3333
8| 0.0000 0.6667	9| 0.3333 0.6667	10| 0.6667 0.6667	11| 1.0000 0.6667
12| 0.0000 1.0000	13| 0.3333 1.0000	14| 0.6667 1.0000	15| 1.0000 1.0000

TexCd for each particle

TexCoords 1

Texture Coordinates 1: we keep the texCoords coming out of the original quad. we'll use these coords to place an image texture onto the particles.
we don't need to resample these Coords because we already did for position and here will be done automatically by the vertex buffer.

0.0000	0.0000
1.0000	0.0000
0.0000	1.0000
1.0000	1.0000

Texture Coordinate 1 XY

Indices

0	1	2
1	3	2

Indices

Indices from Quad mesh

to build the first face that compounds the quad the GPU takes in order vertex 0, 1 and 2.

to build the second face GPU takes in order vertex 1, 3 and 2.

We need to repeat this construction scheme for all the quads of the Mega Mesh.

Quad mesh rappresentation

indices count per quad

final indices count

0	1	2
1	3	2
0	1	2
1	3	2
0	1	2
1	3	2
0	1	2
1	3	2

We can't just repeat these indices (like we did for XYZ position): the second quad (particle) won't be composed by vertex 0,1,2,3! it must use vertices 4,5,6,7.

Here we generate the offset spread to add to the original index construction scheme. In this way all the quad will use their relative points to build faces

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

vertex count per quad

0	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60

vertex index offset for each quad

resample the offset 6 times for all 6 indices of each quad

0	1	2	Particle 00
1	3	2	
4	5	6	Particle 01
5	7	6	
8	9	10	Particle 02
9	11	10	
12	13	14	Particle 03
13	15	14	
...	...	...	...
...	...	...	

0	0	0
0	0	0
4	4	4
4	4	4
8	8	8
8	8	8
12	12	12
12	12	12

Resampled offset

0	1	2

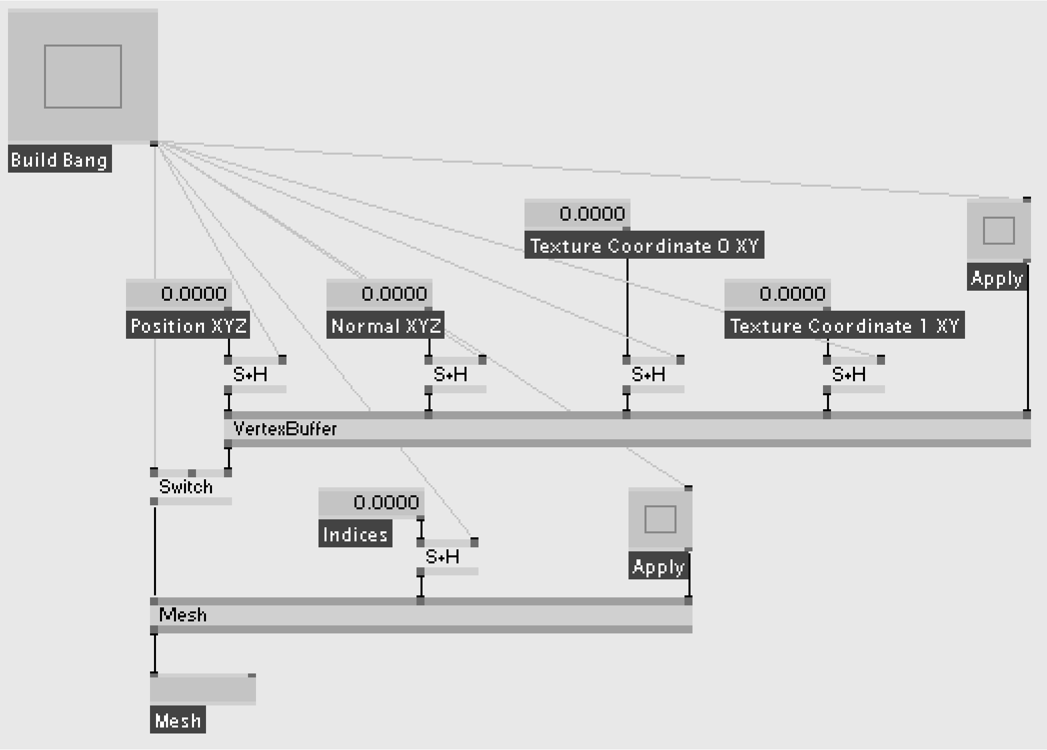
Indices

Couple of things on Mega Mesh:

- When you have thousands of particles, the spreads inside Mega Mesh Module will have a huge amount of slices (up to millions!).

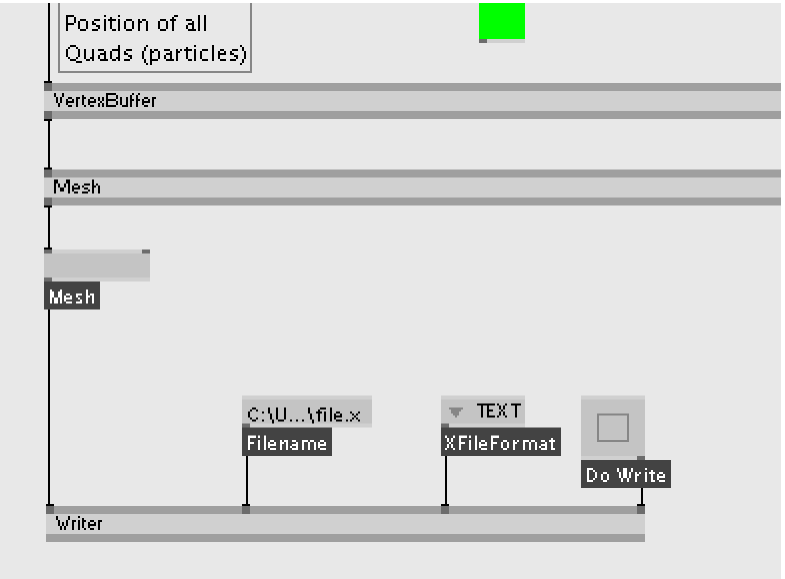
  For this reason you need to **optimize** the module and switch all the slices off when the mesh is builden (you build the mesh just one time, not every frame).

  Use some ***S+H*** nodes just before the *VertexBuffer* inputs and remember to "**Apply**" both *VertexBuffer* and *Mesh* nodes only when generating the mesh.



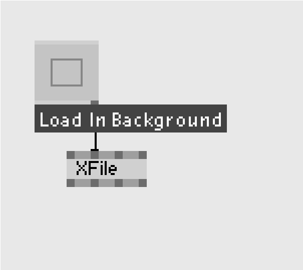
- When you find the correct settings for your mesh you can write it as an X File (Mesh file with extension .x) using *Writer (EX9. Geometry Xfile)* node.

...(C1, G1, D2, Eb2, G1, D2, Eb2, G1, D2, Eb2)...



Then you can use *FileX (EX9. Geometry Load)* node to load the mesh instead of using the Mega Mesh Module.

The cool thing is the "*Load In Background*" pin. If Enabled you can upload huge meshes without freeze the framerate. Quite amazing...



....ok, from now on it's really problematic to work without GPU... ehehehe

I'll post this first part of the Document so you can start to look at it.

As soon as possible I'll continue...

Keep updated

Natan

...!!!...   ....It seems will be a loooong paper in the end... :)