Using GPU Shaders for Visualization, III

Mike Bailey, Oregon State University
mjb@cs.oregonstate.edu

Introduction

GPU Shaders are not just for glossy special effects. In [Bailey2009] and [Bailey2011], we looked at some uses for GPU shaders in visualization. In this article, we continue with that pattern by covering two of the newest features of OpenGL – compute shaders and shader storage buffer objects, which were just announced last summer as part of OpenGL 4.3.

Originally, OpenGL was for graphics only. But, it wasn't long until practitioners were gazing longingly at the power on the GPU and wanting to use it for non-graphics data-parallel computing. This created the field known as General Purpose GPU (GPGPU) [Owens2007]. This was quite effective, and some amazing results were achieved, but it was still an awkward workaround requiring the data-parallel problem to be recast as a pixel-parallel problem first.

True mainstream general-purpose programming on GPUs emerged with NVIDIA's Compute Unified Device Architecture (CUDA) [Kirk2010]. This treated the GPU as a real compute engine without any need to include graphics remnants. Later, OpenCL [Gaster2012, Munshi2012] was developed to create a multi-vendor GPU-programming standard.

OpenGL 4.3 also introduced shader storage buffer objects which support the compute shaders, and make it much easier to get data into and out of them. This means that, finally, using the GPU for data-parallel visualization computing is a first-class feature of OpenGL. This will have a very positive impact on real-time visualization techniques such as particle advection, isosurfaces, etc.

Introduction to OpenGL Compute Shaders

Using compute shaders looks very much like a standard two-pass rendering solution. The GPU gets invoked twice, once for the compute operation and once (or more) for the graphics operation. The compute shader manipulates GPU-based data. The OpenGL rendering pipeline creates a scene based on those new data values. This process is shown in Figure 1.

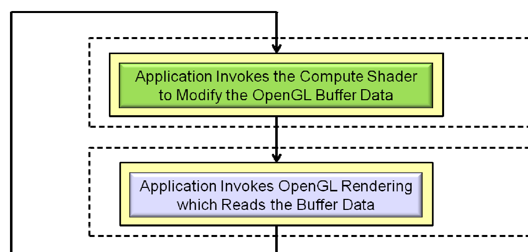


Figure 1. The compute shader paradigm involves round-robin execution between the compute and rendering pieces of the application.
I'LL SEND YOU A HIGH-RES VERSION OF THIS

At this point, we will start diving down into details, and will assume knowledge of OpenGL and GLSL shaders, including how to write them and how to compile and link them.[1]

[1] For OpenGL background, see [Angel2011]. For GLSL shader background, see [Rost2009] and [Bailey2012]

An OpenGL compute shader is a single-stage GLSL program that has no direct role in the graphics rendering pipeline. A compute shader sits outside the pipeline and manipulates data that it finds in the OpenGL buffers. With the exception of a handful of dedicated built-in GLSL variables, compute shaders look identical to all other GLSL shader types. The programming syntax is the same. They have access to the same data that is found in the OpenGL readable data types, such as textures, buffers, image textures, atomic counters, etc. Their outputs are any of the same OpenGL writeable data types, such as some buffer types, image textures, atomic counters, etc. But, they have no previous-pipeline-stage inputs nor next-pipeline-stage outputs because, to them, there is no pipeline and there are no other stages.

Comparison with OpenCL

In many ways, GLSL compute shaders look a lot like OpenCL programs. Both manipulate GPU-based data in a round-robin fashion with the rendering. The programming languages look similar.[2] But, there are some important differences:

- OpenCL is its own entity. Using OpenCL requires a several-step setup process in the application program.
- OpenCL requires separate drivers and libraries.
- Compute shaders use the same context as OpenGL rendering. OpenCL requires a context switch before and after invoking its data-parallel compute functions.
- OpenCL has more extensive computational support in its language.

In summary, it appears that OpenCL should continue to be used for large GPU data-parallel computing applications. But, for many simpler applications, compute shaders will slide more easily into your existing shader-based program.

What Is Different about Using a Compute Shader Compared with any other Shader Type?

For the most part, writing and using a compute shader looks and feels like writing and using any other GLSL shader type, with these exceptions:

- The compute shader program must have no other shader types in it.
- When creating a compute shader, use GL_COMPUTE_SHADER as the shader type in the glCreateShader( ) function call.
- A compute shader has no concept of in or out variables.
- A compute shader must declare the number of work-items in each of its work-groups in a special GLSL layout statement.

This last bullet is worth further discussion. In version 3, GLSL introduced the *layout* qualifier to tell the GLSL compiler and linker about the storage of your data. It has been used for such things as telling geometry shaders what their input and output topology types are, what symbol table locations certain variables will occupy, and the binding points of indexed buffers. In OpenGL 4.3, the use of the layout qualifier has been expanded to declare the local data dimensions, like this:

```
layout( local_size_x = 128 ) in;
```

More will be covered on this later.

Shader Storage Buffer Objects

Oftentimes the tricky part of using GLSL shaders for visualization is getting large amounts of data in and out of them. OpenGL has created several ways of doing this over the years, but each seems to have had something about it that made it

[2] Most of the differences in language are superficial, such as OpenCL using SIMD variables named float[2-4] and GLSL using vec[2-4].

cumbersome for visualization use. For example, textures and uniform buffer objects can only be read from, not written back to. Image textures can be both read and written, but are backed by textures, which are not as data-flexible as buffer objects.

In a CPU-only data-parallel application, oftentimes the most convenient data structure is an array of structures, where each element of the array holds one instance of all the data variables. But, none of these GLSL storage methods have allowed that familiar storage scheme to be used in shader programming.

The new shader storage buffer object (SSBO) was created to fix all that. SSBOs cleanly map to arrays of structures, which make them convenient and familiar for data-parallel computing. Rather than talk about them, it is easiest to show their use in actual code. The following example shows a simple particle system, which uses both SSBOs and compute shaders. Listing 1 shows the CPU code being used to setup the required position and velocity SSBOs.

Some things to note in the code:

- Generating and binding a shader storage buffer object happens the same as any other buffer object type, except for its GL_SHADER_STORAGE_BUFFER identifier.
- These SSBOs are specified with NULL as the data pointer. The data could have been filled here from pre-created arrays of data, but oftentimes it is more convenient to create the data on the fly and fill the buffers rather than allocating large arrays first. So, in this case, data values are filled a moment later using buffer-mapping.
- The glBufferData( ) call shown here uses the hint GL_STATIC_DRAW. The OpenGL books all say to "use GL_DYNAMIC_DRAW when the values in the buffer will be changed often." With compute shaders, that phrase is now incomplete. It needs to be changed to say to "use GL_DYNAMIC_DRAW when the values in the buffer will be changed often *from the CPU.*" When the data values will be changed from the *GPU*, GL_STATIC_DRAW causes them to be kept in GPU memory, which is what we want.
- The expected call to glMapBuffer( ) has been replaced with a call to glMapBufferRange( ), which allows a parameter specifying that the buffer will be entirely discarded and replaced, thus hinting to the driver that is should remain in the memory (GPU) where it currently lives.
- The calls to glBindBufferBase( ) allow these buffers to be indexed, meaning that they are assigned integer indices which can be referenced from the shader using a layout qualifier.

Listing 2 shows how the compute shader accesses the SSBOs. Some things to note:

- The shader code uses the same set of structures to access the data as the C code did, with the data types changed to match GLSL syntax.
- The SSBO layout statements provide the binding indices so that the shader knows which SSBO to point to.
- The open brackets in the SSBO layout statement show the new GLSL syntax to define an array of structures. There can actually be more items in the definition than just that open-bracketed array, but it is required that the open-bracketed item be the *final* variable in the list.

Calling the Compute Shader

A compute shader is invoked from the application with the following function:

```
glDispatchCompute( numgx, numgy, numgz );
```

where the arguments are the number of work groups in x, y, and z, respectively. A compute shader expects you to treat your data parallel problem as a 3D array of work

groups to process.[3] The grid of work groups is shown in Figure 2.

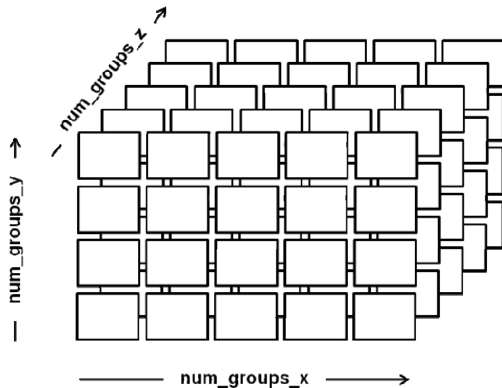


Figure 2. The 3D grid of work groups allows the data to be dimensioned in a way that is convenient to the application. I'LL GET YOU A HIGH-RES VERSION OF THIS

Each work group consists of some number of work items to process. The number of work groups times the number of work items per work group gives the total number of elements that are being computed. How you divide your problem into work groups is up to you, however it is important to experiment with this as some combinations will starve the GPU processors of work to do.[4] We did experiment with the local work group size for one particular application. The results are coming up in Figure 5.

Compute Shader Built-in Variables

GLSL compute shaders have several built-in variables. These are not accessible from any other shader types:

```
in   uvec3       gl_NumWorkGroups ;
const  uvec3     gl_WorkGroupSize ;
in   uvec3       gl_WorkGroupID ;
in   uvec3       gl_LocalInvocationID ;
in   uvec3       gl_GlobalInvocationID ;
in   uint        gl_LocalInvocationIndex ;
```

[3] Although, for a 2D problem, the numgz will be 1, and for a 1D problem, numgx and numgy will both be 1.
[4] Also, there are some OpenGL driver-imposed limits on the number of work groups and on the work group size.

gl_NumWorkGroups are the number of work groups in all three dimensions. They are the same numbers as you used in the glDispatchCompute( ) call.

gl_WorkGroupSize are the sizes of the work groups in all three dimensions. They are the same numbers you used in the layout call.

gl_WorkGroupID are the work group numbers in all three dimensions that the current instantiation of the computer shader is in.

gl_LocalInvocationID is where, in all three dimensions, the current instantiation of the compute shader is inside its own work group.

gl_GlobalInvocationID is where, in all three dimensions, the current instantiation of the compute shader is within all instantiations.

gl_LocalInvocationIndex is a 1D abstraction of gl_LocalInvocationID. It is used to allocate work-group shared data arrays (which we aren't covering here).

These built-in variables have the following size ranges:

0 ≤ gl_WorkGroupID ≤ gl_NumWorkGroups – 1

0 ≤ gl_LocalInvocationID ≤ gl_WorkGroupSize – 1

gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID

gl_LocalInvocationIndex = gl_LocalInvocationID.z * gl_WorkGroupSize.y * gl_WorkGroupSize.x + gl_LocalInvocationID.y * gl_WorkGroupSize.x + gl_LocalInvocationID.x

The Particle System Physics

Listing 4 shows the code to perform the particle system physics:

- A layout statement declares the work group size to be 128x1x1

- Gravity (G) and the time step (DT) are defined. G is a vec3 so that it can be used in a single line of code to produce the particle's next position.
- This code runs once for each particle. The variable `gid` is that particle's number in the entire list of particles. `gid` indexes into the array of structures.

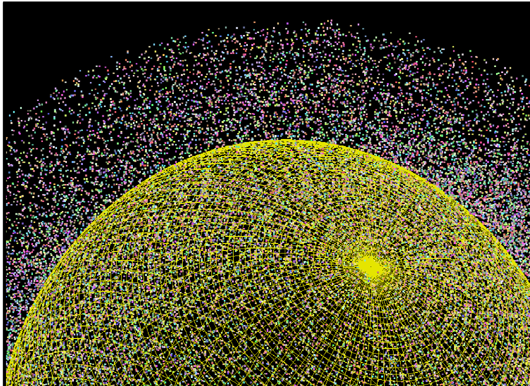


Figure 3. The particle system updates at a rate of 1.3 gigaparticles per second. I'LL SEND YOU A HIGH-RES VERSIUON OF THIS

Particle Advection

Listing 5 shows how the particle system is turned into a first-order visualization particle advection, being fed by a velocity equation as a function of particle location.

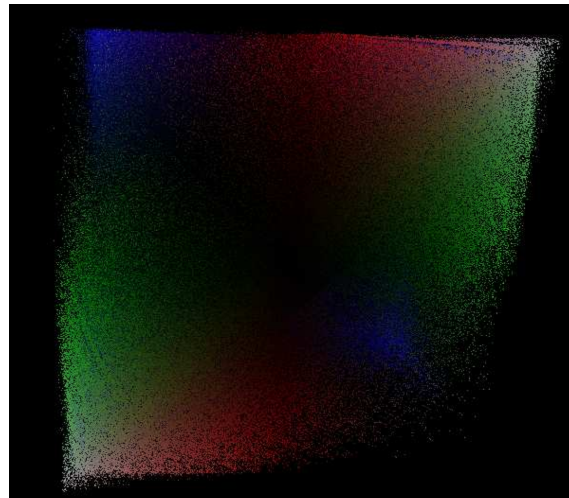
Notice that:

- The code uses #defines to simulate typedefs. GLSL does not (yet) support typedefs, which, I think, impacts the readability of the code. In this case, even though both points and velocities are really vec3s, it helps the code's readability when one can distinguish a coordinate from a vector.
- This code runs once for each particle. The variable `gid` is the global ID, that is, the particle's number in the entire list of particles. `gid` indexes into the array of structures.
- The Velocity( ) function computes (vx,vy,vz) as a function of position.
- The equation is defined for x,y,z between -1. and 1. If a point has moved out of bounds, it is reset to its original position.
- The line `pp = p + DT * vel;` performs a first-order particle step.
- The particle's color is set from the three velocity components, as a way to keep track of each particle direction. It could easily be colored to show other quantities.

However, most 3D flow field data is not given as an equation. Listing 6 shows how one would hide the velocity field values in a 3D texture, using the r, g, and b texture components for the x, y, and z velocity components [Bailey2011]. The only trick is that the position must be converted from its coordinate values in the range [-1.,1.] to the texture lookup range of [0.,1.]. The code line

```
vec3 stp = ( pos + 1. ) / 2.;
```

does that for us.



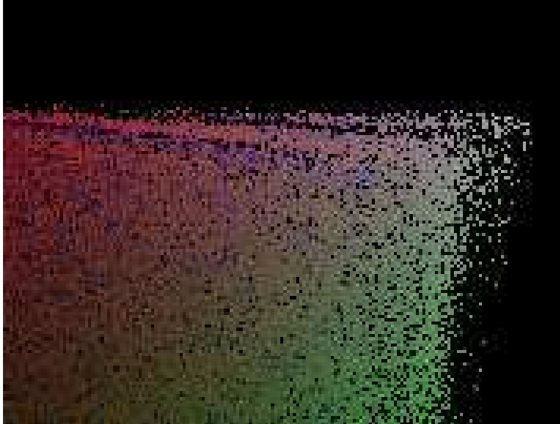


Figure 4. A visualization particle advection is like the particle system, but includes a velocity vector lookup.
I'LL SEND YOU A HIGH-RES VERSION OF THIS

Performance

The code was tested with 1,048,576 (1024^2) particles on an NVDIA GeForce 480. Different work group sizes were tested. Figure 5 shows the compute speeds, measured in GigaParticles/Second.

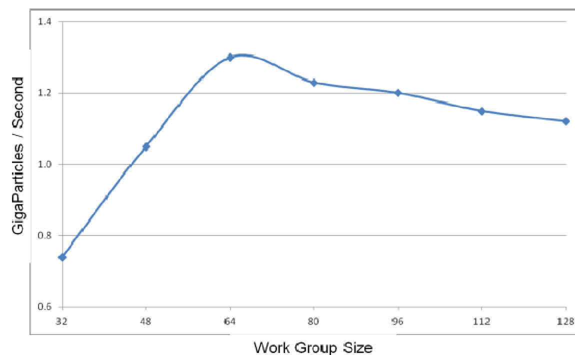


Figure 5. This graph shows the compute shader performance as a function of work group size.
I'LL SEND YOU A HIGH-RES VERSIUON OF THIS

The top performance was 1.3 gigaparticles/sec resulting from a work group size of 64. On the 480, each Streaming Multiprocessor has 32 SIMD units to perform the data parallel particle advection. No work group size smaller than 32 would make sense, as it would leave some of those units unused. By using a work group size of exactly 32, however, anytime the execution blocks (for a memory access, for instance) the SIMD units would have nothing to do. Thus, it makes sense that a work group size of 64 or more would produce better performance than 32. Figure 5 indicates that, for this application at least, beyond 64 doesn't help and even hurts some. This is application-dependent. Different shader applications behave differently, and so it is best to benchmark rather than assume.

Conclusions

After many years of using the GPU for data-parallel visualization computing by employing various hacks and workarounds, it is a relief to finally have all of the pieces become first-class citizens of the GLSL shader language. Now we can realize the full potential of GPGPU using C-looking arrays of structures and fine-grained data parallel computing, all within the comfortable confines of OpenGL.

We do have to change some of our rules of thumb, however. The OpenGL books all say to "use GL_DYNAMIC_DRAW when the values in the buffer will be changed often" . That now needs to be updated to say to "use GL_DYNAMIC_DRAW when the values in the buffer will be changed often *from the CPU*."

Acknowledgements

Many thanks to NVIDIA and Intel for their continued support of our shader research. Also, much gratitude to Piers Daniell of NVIDIA for his help and patience as we were first working with compute shaders.

References

Angel2011
Edward Angel and Dave Shreiner, *Interactive Computer Graphics: A Top-down Approach with OpenGL,* 6th Edition, Addison-Wesley, 2011.

Bailey2009
Mike Bailey, "Using GPU Shaders for Visualization", *IEEE Computer Graphics and Applications*, Volume 29, Number 5, 2009, pp. 96-100.

Bailey2011
Mike Bailey, "Using GPU Shaders for Visualization, II", *IEEE Computer Graphics and Applications*, Volume 31, Number 2, 2011, pp. 67-73.

Bailey2012
Mike Bailey and Steve Cunningham, *Graphics Shaders: Theory and Practice*, Second Edition, Taylor Francis, 2012.

Gaster2012
Benedict Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa, *Heterogeneous Computing with OpenCL*, Morgan-Kaufmann, 2012.

Kirk2010
David Kirk, Wen-mei Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan-Kaufmann, 2010.

Munshi2012
Aaftah Munshi, Benedict Gaster, Timothy Mattson, James Fung, and Dan Ginsburg, *OpenCL Programming Guide* Addison-Wesley, 2012.

Nichols1998
Bradford Nichols, Dick Buttlar, and Jacqueline Proudx Farrell, *Pthreads Programming*, O'Reilly, 1998.

Owens2007
John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy Purcell, "A Survey of General-Purpose Computation on Graphics Hardware", *Computer Graphics Forum*, 26(1), March 2007, pp. 80-113.

Rost2009
Randi Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen, *OpenGL Shading Language*, Addison-Wesley, 2009.

Listing 1. Allocating and Filling the Shader Storage Buffer Objects

```
#define NUM_PARTICLES      1024*1024  // 1M particles to move
#define WORK_GROUP_SIZE    128        // # work-items per work-group

struct pos
{
      float x, y, z, w; // positions
};

struct vel
{
      float vx, vy, vz, vw;   // velocities
};

// need to do this for both position and velocity of the particles:

GLuint  posSSbo;
GLuint  velSSbo

glGenBuffers( 1, &posSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, posSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER,
              NUM_PARTICLES * sizeof(struct pos),
              NULL, GL_STATIC_DRAW );
GLint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT ;
            // the invalidate makes a big difference when re-writing
struct pos *points = (struct pos *) glMapBufferRange(
            GL_SHADER_STORAGE_BUFFER, 0,
            NUM_PARTICLES * sizeof(struct pos), bufMask );

for( int i = 0; i < NUM_PARTICLES; i++ )
{
      points[ i ].x = Ranf( XMIN, XMAX );
      points[ i ].y = Ranf( YMIN, YMAX );
      points[ i ].z = Ranf( ZMIN, ZMAX );
      points[ i ].w = 1.;
}
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );


glGenBuffers( 1, &velSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, velSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER,
              NUM_PARTICLES * sizeof(struct vel),
              NULL, GL_STATIC_DRAW );
struct vel *vels = (struct vel *) glMapBufferRange(
            GL_SHADER_STORAGE_BUFFER, 0,
            NUM_PARTICLES * sizeof(struct vel), bufMask );
```

```
for( int i = 0; i < NUM_PARTICLES; i++ )
{
      vels[ i ].vx = Ranf( VXMIN, VXMAX );
      vels[ i ].vy = Ranf( VYMIN, VYMAX );
      vels[ i ].vz = Ranf( VZMIN, VZMAX );
      vels[ i ].vw = 0.;
}
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );


glBindBufferBase( GL_SHADER_STORAGE_BUFFER,  4,  posSSbo );
glBindBufferBase( GL_SHADER_STORAGE_BUFFER,  5,  velSSbo  );
```

Listing 2. How the Shader Storage Buffer Objects Look in the Shader

```
#version 430 compatibility
#extension GL_ARB_compute_shader :                  enable
#extension GL_ARB_shader_storage_buffer_object :    enable;

struct pos
{
      vec4 pxyzw;       // positions
};

struct vel
{
      vec4 vxyzw;       // velocities
};

layout( std140, binding=4 )  buffer  Pos  {
      struct  pos  Positions[  ];         // array of structures
};

layout( std140, binding=5 )  buffer  Vel  {
      struct  vel  Velocities[  ];        // array of structures
};
```

Listing 3. Invoking the Compute Shader

```
glUseProgram( MyComputeShaderProgram );
glDispatchCompute( NUM_PARTICLES  / WORK_GROUP_SIZE, 1,  1 );
glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );

      . . .

glUseProgram( MyRenderingShaderProgram );

// render the scene
```

Listing 4. Shader Code for One Particle

```
layout( local_size_x = 128,  local_size_y = 1, local_size_z = 1 )   in;

const  vec3   G     =  vec3( 0., -9.8, 0. );
const  float  DT    =  0.1;
      . . .
uint gid = gl_GlobalInvocationID.x;       // the .y and .z are both 1
vec3 p   = Positions[  gid  ].pxyzw.xyz;
vec3 v   = Velocities[ gid  ].vxyzw.xyz;
vec3  pp = p + v*DT + .5*DT*DT*G;
vec3  vp = v + G*DT;
Positions[   gid  ].pxyzw.xyz  = pp;
Velocities[  gid  ].vxyzw.xyz  = vp;
```

Listing 5. Visualization First-order Particle Advection using a Velocity Equation

```
// roll your own typedefs:

#define point       vec3
#define velocity    vec3


velocity
Velocity( point pos )
{
    // -1. <= pos.x,y,z <= +1
    float x = pos.x;
    float y = pos.y;
    float z = pos.z;
    return velocity(
            y * z * ( y*y + z*z ),
            x * z * ( x*x + z*z ),
            x * y * ( x*x + y*y )
      );
}

void
main( )
{
    uint  gid = gl_GlobalInvocationID.x;

    point p  = Positions[ gid ].pxyzw.xyz;
    point pp = p;

    if( any(  greaterThan(  abs(p.xyz), point(1.,1.,1.)  )  )  )
    {
        pp = OrigPositions[gid].pxyzw.xyz;
        Colors[ gid ].crgba     = vec4( 0., 0., 0., 1. );
    }
    else
    {
        velocity vel = Velocity( p );
        pp = p + DT * vel;
        Colors[  gid  ].crgba     = vec4( abs(vel)/3., 1. );
    }

    Positions[ gid ].pxyzw.xyz = pp;
}
```

Listing 6. Visualization First-order Particle Advection using a Texture Velocity Field

```
velocity
Velocity( vec3 pos )
{
    // -1. <= pos.x,y,z <= +1
    vec3 stp = ( pos + 1. ) / 2.;

    // 0. <= s,t,p <= 1.
    return texture( velocityTexture, stp ).rgb;
}
```