Visualizing Complex Functions Using GPUs

Khaldoon Ghanem

RWTH Aachen University
German Research School for Simulation Sciences
Wilhelm-Johnen-Straße
52428 Jülich

E-mail: k.ghanem@grs-sim.de

Abstract:
This document explains some common methods of visualizing complex functions and how to implement them on the GPU. Using the fragment shader, we visualize complex functions in the complex plane with the domain coloring method. Then using the vertex shader, we visualize complex functions defined on a unit sphere like spherical harmonics. Finally, we redesign the marching tetrahedra algorithm to work on the GPGPU frameworks and use it for visualizing complex scaler fields in 3D space.

1 Introduction

GPUs are becoming more and more attractive for solving computationally demanding problems. This is because they are cheaper than CPUs in two senses. First, they provide more performance for less cost .i.e. cheaper GFLOPs. Second, they are more energy efficient .i.e. cheaper running times. This reduced cost comes at the expense of less general purpose architecture and hence a different programming model.

There are two main ways of programming GPUs. The first one is used in the graphics community using shading languages like HLSL, GLSL and Cg. In these languages, the programmer deals with vertices and fragments and processes them with the so called vertex and fragment shaders, respectively. Actually, this has been the only way of programming GPUs for a while. Fortunately, in the recent years, frameworks for general programming have been developed like CUDA and OpenCL. They are much more suited for expressing the problem more abstractly in terms of threads. These threads are then processed with the so called kernels.

Visualizing complex functions makes an ideal problem to be solved on the GPU because the function needs to be evaluated at different points of the domain and these points are processed independently. We deal in this document with three types of complex functions; each one requires different visualization method and each method is most appropriately implemented on the GPU in a different way. The complex functions, we are addressing, are functions in complex plane, functions on unit sphere and functions in 3D space.

2 Complex Functions in Complex Plane

To visualize complex functions of a single complex variable, we would need a four-dimensional space! However, by encoding the values in the complex plane in colors, we are able to visualize these functions in two dimensions. This method is called **Domain Coloring** [1].

2.1 Domain Coloring

For visualizing the function $f : \mathbb{C} \to \mathbb{C} : w = f(z)$:

- Cover the range complex plane with some color map or scheme. i.e give each point w a color.
- For each point of the domain complex plane z, compute $w = f(z)$ and then color z with the color of w.

The result of the above procedure is a colored image of the domain (see figure 1).

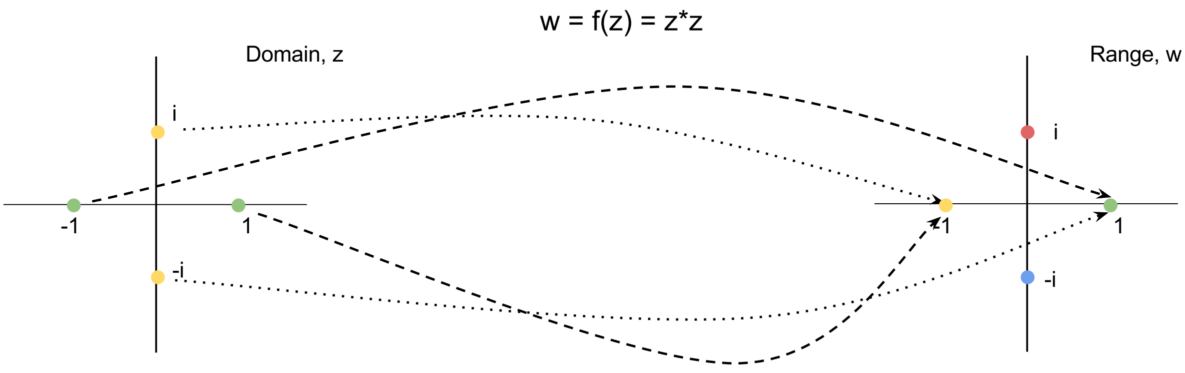

Figure 1: As an illustrating simple example, let us assume we want to visualize the square function z^2 at only four points $1, -1, i, -i$. We give each of these points in **Range** plane a unique color. Then we compute the function values at these points and color them according to the result: $f(1) = f(-1) = 1$ (green), and $f(i) = f(-i) = -1$ (yellow). The image to the right is called *color map* and the one to the left is *plot of* z^2

2.2 Choosing Color Map

The goal here is to give each complex number a color. In principle, you can use any picture to cover the complex plane. However, the resulting function plots are usually not easy to interpret. Also, the picture cannot cover the whole infinite plane, and thus we need a more systematic way.
For a complex number w, choose color hue according to the argument $\arg(w)$ from a smooth color sequence (color gradient). Then, choose color brightness according to the fractional part of the $\log_2 |w|$.(see figure 2). For a through explanation on how to read the plots of such color maps, see [2].

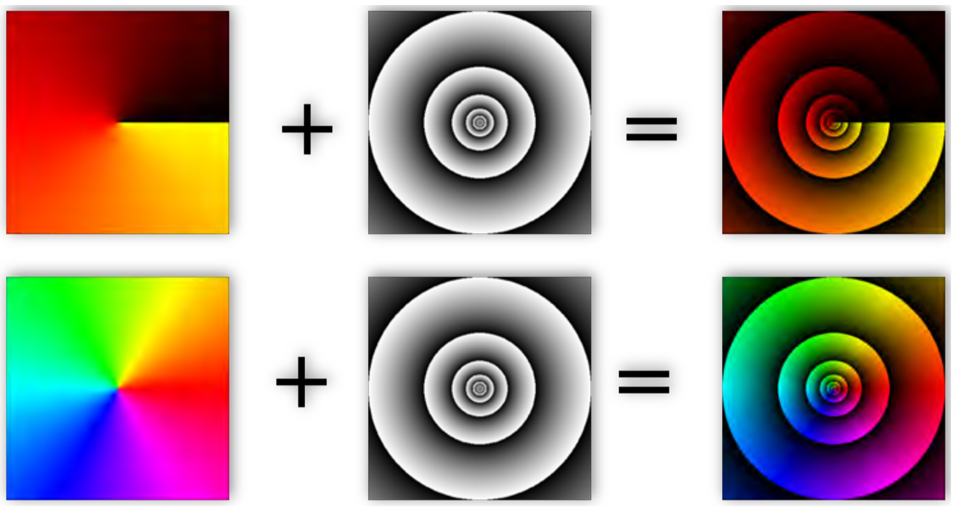

Figure 2: Two color maps differ by argument encoding. The first map interpolate between three colors; black, red and yellow. The second map spans the whole spectrum.

2.3 Using the GPU

The coloring of the domain complex plane is embarrassingly parallel as each point of the plane is processed independently. To utilize the GPU in this problem, we map complex points to screen pixels and color them with a fragment shader written in some shading language. The procedure outline is as following:

- Make a rectangle covering the whole screen using the graphics API , OpenGL or DirectX.
- The rectangle's fragments will be created by the fixed functionality of the graphics pipeline. These fragments are the final pixels of the screen, because there is only one primitive and it is not clipped.
- Fragment shader will be executed for each fragment. Inside fragment shader:
  - Transform the fragment screen coordinates to a complex point z using a transformation matrix passed from main program.
  - Compute the function at that point $f(z) = w$.
  - Determine the fragment color according to the color map at the resulting point w.

2.4 Implementation

We developed an application that visualize complex functions of a single variable. The main program is written using OpenGL graphics API and GLUT library while the fragment shader is written in GLSL shading language. The application takes as input a list of complex functions expressed with common mathematical symbols. Function expressions are parsed and translated into appropriate shader calls and operations using a lexical analyzer and a parser generated by lex and yacc tools.

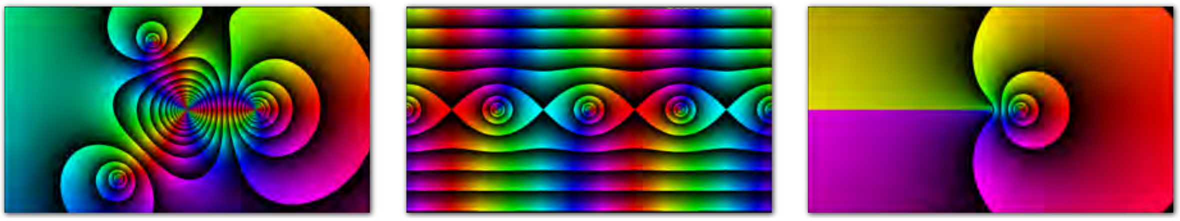

Figure 3: Plots of some complex functions generated using our domain coloring program: $(z-2)^2(z+1-2i)(z+2+2i)/z^3$ (left), $sin(z)$ (center), $log(z)$ (right).

3 Complex Functions on Unit Sphere

The second class of complex functions we address is functions defined on a unit sphere $f : [0,\pi] \times [0, 2\pi) \to \mathbb{C} : f(\theta,\varphi) = w$. Although we could apply the domain coloring method to the surface of unit sphere, we are already visualizing in three dimensions and it is convenient to use the extra dimension we have at our disposal.

- Start with a unit sphere in 3D space.
- Each point on the surface has the spherical coordinates (r, θ, φ).
- Deform the sphere such that for each point $r = |f(\theta,\varphi)|$.
- Color the surface according to $\arg(f(\theta,\varphi))$ using some smooth color sequence as we did in domain coloring method 2.1.

3.1 Calculating the normals

Since we are now working in 3D, normals at surface points should be provided for appropriate lightening. To calculate them, we express the deformed unit sphere as an isosurface of a scalar field: $F(r,\theta,\varphi) = \sqrt{f(\theta,\varphi)\overline{f}(\theta,\varphi)} - r$ with an isovalue equals zero. Then the gradient of the field ∇F is normal to the isosurface.
The gradient in spherical coordinates is calculated using

$$\nabla F = \frac{\partial F}{\partial r}\hat{\boldsymbol{r}} + \frac{1}{r}\frac{\partial F}{\partial \theta}\hat{\boldsymbol{\theta}} + \frac{1}{r\sin\theta}\frac{\partial F}{\partial \varphi}\hat{\boldsymbol{\varphi}}$$

where

$$\frac{\partial F}{\partial r} = -1 \qquad \frac{\partial F}{\partial \theta} = \frac{\frac{\partial \overline{f}}{\partial \theta} f + \frac{\partial f}{\partial \theta}\overline{f}}{2\sqrt{f\overline{f}}} = \frac{Re[\frac{\partial \overline{f}}{\partial \theta} f]}{|f|} \qquad \frac{\partial F}{\partial \varphi} = \frac{Re[\frac{\partial \overline{f}}{\partial \varphi} f]}{|f|}$$

So we don't only need to calculate the function value but also the partial derivatives of its complex conjugate.

3.2 Using the GPU

We generate a unit sphere (vertices and triangles). Then, using a vertex shader, each vertex of the sphere is modified independently. Inside the vertex shader:

- Retrieve vertex's angles (θ, φ)
- Compute function value $f(\theta, \varphi)$
- Modify vertex coordinates such that $r = |f(\theta, \varphi)|$
- Modify vertex color according to $\arg(f(\theta, \varphi))$ using some smooth color sequence.
- Compute partial derivatives of the function and use them to compute the gradient vector.
- Modify vertex normal such that it points in the direction of the gradient.

3.3 Implementation

Spherical harmonics are the most well known complex functions on a unit sphere. They form a complete set of orthonormal functions thus any square-integrable complex function on a unit sphere can be expanded as linear combination of them. Due to their importance, we developed an application for visualizing linear combinations of spherical harmonics. The main program is written using OpenGL graphics API and GLUT library, while the vertex shader is written in GLSL shading language. For a stable method of evaluating spherical harmonics see [3].

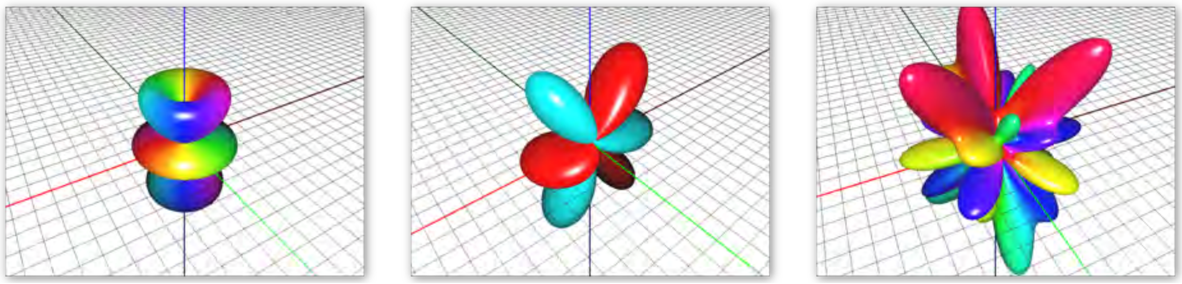

Figure 4: Plots of spherical harmonics Y_3^1(left), its real part (center), and the linear combination $\frac{1}{2}Y_1^0 + iY_5^3 + (\frac{1}{2} + i\frac{1}{2})Y_7^{-3}$ (right)

4 Complex Functions in 3D

We often need to visualize discrete complex functions defined in 3D space $f : \mathbb{R}^3 \to \mathbb{C}$ like wave functions resulting form quantum mechanics calculations. First, we use the trick of encoding the argument of the complex function in color as in the domain coloring method 2.1. What is left then, is visualising the absolute value which can be considered as a scalar field in 3D space. This suggests the following procedure:

- Specify one absolute value to visualize.
- Calculate an isosurface of the absolute value using marching tetrahedra.
- Color the isosurface according to the argument using some smooth color sequence.
- If necessary, change the isovalue and repeat process to gain more info.

4.1 Marching Tetrahedra

Marching Tetrahedra is an isosurface extraction algorithm. Given a 3D scalar field, it finds the surface on which the field has a constant value, the isovalue. The key idea of marching tetrahedra is noting that isosurface of a volume is the union of the isosurfaces of its components.

The field values are given at the points of a mesh. Any mesh is naturally divided into mesh cells. In this document, we consider structured meshes with *parallelepiped* cells (usually cubes). We could take the mesh cell as our building block and find the isosurface of each mesh cell independently and then collect them to form the final isosurface. This would be called **Marching Cubes Algorithm**[4, 5]. Marching cubes suffer from some ambiguities in finding the isosurface of a mesh cell. These ambiguities are not present in marching tetrahedra.

In **Marching Tetrahedra Algorithm**[6, 7], we go one step further beyond marching cubes and divide each mesh cell into six tetrahedra. Then our building block is the tetrahedron. There are several ways of splitting a hexhedron (usually a cube) into six tetrahedra. The way we do it is illustrated in (figure 5).

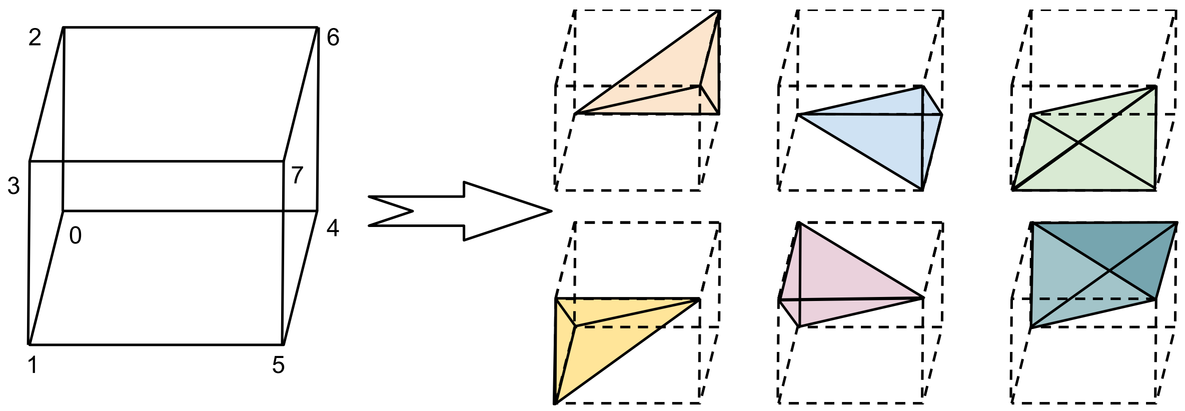

Figure 5: Splitting a cube (or a parallelepiped in general) into six tetrahedra. They are listed from right to left, top to bottom: (0,6,4,7), (0,4,5,7), (0,5,1,7), (0,5,1,7), (0,1,3,7), (0,3,2,7), (0,2,6,7).

To find the isosurface of one tetrahedron, we check whether each of its four corners is above or below the isovalue (let's denote them as a plus or minus, respectively). The isosurface clearly passes only through edges connecting corners of opposite signs. Since we only know field values at corners, we linearly interpolate them along the edges to get the intersection points (see figure 6).

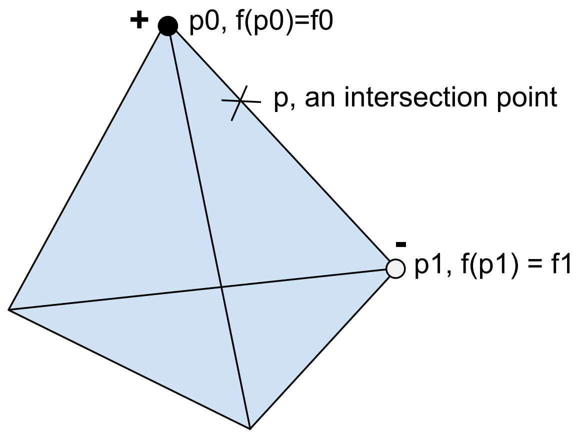

Figure 6: The intersection of isosurface with an edge can be computed by considering a linear interpolation along the edge. If two corners p0 and p1 are of opposite signs then the intersection point on their edge p is computed as $p = \alpha p_1 + (1 - \alpha)p_0$ where $\alpha = \frac{c - f_0}{f_1 - f_0}$ and c is the isovalue.

Besides getting the intersection points (isosurface vertices), we want to connect them. i.e form triangles. First, note that the triangulation depends only on the signs of corners, not their exact field values. Since we have four corners with two possible signs each, we have in total $2^4 = 16$ different possible patterns of a tetrahedron.

We enumerate the corners and represent the pattern of a tetrahedron by a 4-bits number, where each bit indicates whether the corresponding corner is above or below the isovalue. Then we enumerate the edges and use a lookup table with 16 entries. Given the tetrahedron's pattern, the table should return isosurface triangles in terms of the edges they connect (see figure 7).

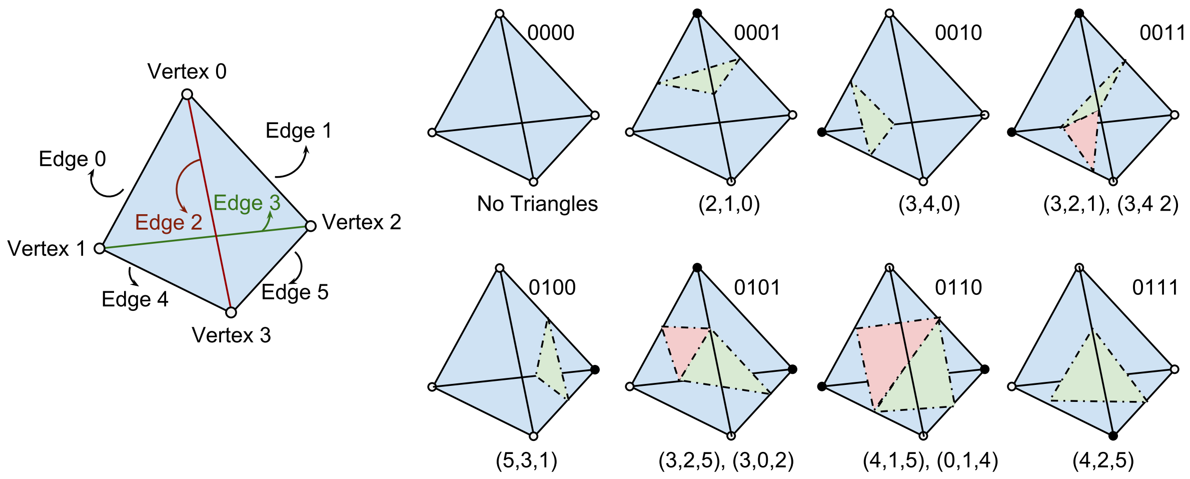

Figure 7: To the left is our choice for enumerating corners and edges. To right is the triangulation for eight different tetrahedron's pattern. The other eight are obtained by flipping patterns' bits and listing edges in reverse order.

4.2 Avoiding Duplicate Isosurface Vertices

We note in the original marching tetrahedra, that isosurface vertices (intersection points) lying on edges shared between adjacent tetrahedra are duplicated. This a disadvantage for two reasons. First, it leads to unnecessary storage of repeated vertices e.g. a vertex on the diagonal of a mesh cell would be repeated six times. Second, if the generated surface is to be processed later or used for some calculations, then this could produce artifacts due to round-off errors. To avoid this duplication, we split the algorithm into two stages: Generating Vertices and Generating Triangles.

In the **Generating Vertices** stage, we loop over all edges. For each edge, check whether it is cut by the isosurface (by checking whether the two end mesh points are of opposite signs). If so, calculate the intersection point (isosurface vertex). Then store the vertex in a hash table indexed by some edge id.

In the **Generating Triangles** stage, we loop over all mesh cells. For each mesh cell, process all six tetrahedra. For each tetrahedron, calculate its pattern. Using the tetrahedron's pattern, get its triangulation. Generate triangles using pointers to vertices (not vertices directly). Get vertices' pointers by looking up edge-vertex hash table generated in the previous stage.

For this to work, we need to identify the edges globally and we do it by associating each mesh point with seven edges (see figure 8). This association is unique and covers all possible edges. Then the id of an edge connecting mesh points (i_1, j_1, k_1) and (i_2, j_2, k_2), where $0 \leq i_2 - i_1 \leq 1$ and $0 \leq j_2 - j_1 \leq 1$ and $0 \leq k_2 - k_1 \leq 1$, is a tuple with two entries. The first is the minimum of the two mesh point ids (i_1, j_1, k_1). The second is a number between 1 and 7 identifying the edge within the mesh point and calculated as $(i_2 - i_1) * 4 + (j_2 - j_1) * 2 + (k_2 - k_1)$.

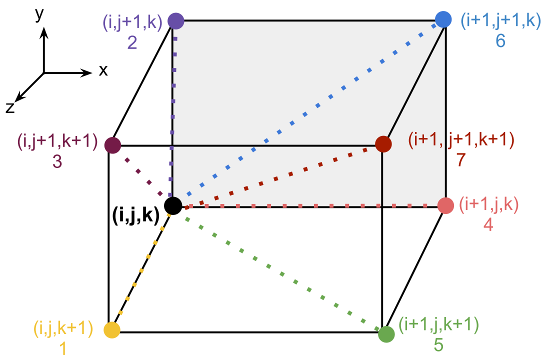

Figure 8: Each mesh point (i,j,k) can be associated uniquely with seven edges. 1: (i,j,k)-(i, j, k+1) ; 2: (i,j,k)-(i, j+1, k); ; 3: (i,j,k)-(i, j+1, k+1) ; 4: (i,j,k)-(i+1, j, k) ; 5: (i,j,k)-(i+1, j, k+1) ; 6: (i,j,k)-(i+1, j+1, k) ; 7: (i,j,k)-(i+1, j+1, k+1).

Note that using this association, the loop over edges in first stage is actually a double loop where the outer is over mesh points and the inner is over edges associated with each mesh point. Also note that the triangulation of a tetrahedron is, as before, expressed in terms of our enumeration of edges inside the tetrahedra (see figure 7). So a mapping from the local edge enumeration to global edge ids should be done before retrieving vertices' pointers form the hash table.

4.3 Using the GPU

Unlike the previous two visualization methods, domain coloring and sphere deformation, we won't use a shading language but rather a GPGPU language like OpenCL or CUDA. Although it is possible to implement the marching tetrahedra on vertex or fragment shaders (see [8, 9, 10]), it is more convenient to express the problem more abstractly using GPGPU. There are implementations of marching cubes on OpenCL and CUDA (see [11, 12]), but to our knowledge, there is no public implementation of marching tetrahedra using GPGPU yet.

We go along the same line of thought as in the last algorithm and separate generating vertices from triangles. First a collection of threads, one for each mesh point, is created. Each of these threads runs a kernel that would generate vertices lying on edges associated with the corresponding mesh point. Then another collection of threads is created, one for each mesh cell. Each of these threads would generate the triangles of tetrahedra associated with the corresponding mesh cell.

There are two main complications on the GPU. First, there is no dynamic memory allocation. So all memory allocation and deallocation should be done before or after the computation but not during it. One solution is to allocate memory for every potential vertex. However, this is impractical, as it would need (7 vertices per mesh point $\times$ 3 coordinates per vertex) = 21 times the storage needed for the scalar field. A similar argument goes for the triangles.
Second, the threads should work independently and write data to distinct memory locations. So each thread should know where to write its results before starting the computation.
Circumventing these problems is done by splitting each stage into yet another three stages. First, we count the number of vertices that would be generated per mesh point. Second, a prefix-scan is done to get the number of generated vertices and a list of addresses for storing the vertices associated with each mesh point. Finally, the necessary memory is allocated and vertices are generated and stored in their appropriate locations. Triangle generation is also split into three stages similarly. This method is highly inspired by [11, 12].

Generating Vertices

1. Allocate necessary memory on GPU for array `mpVertsNum`.
   `mpVertsNum`: An array of bytes. Its size equals the total number of mesh points. Element `mpVertsNum[mp]` contains the number of vertices associated with mesh point `mp`.
2. Run kernel `processMP` for each mesh point to fill `mpVertsNum`.
   Elements of `mpVertsNum` are calculated by counting the number of intersected edges associated with each mesh point. An edge is intersected if its two ends are of different signs (one above the isovalue and the other below).(see figure 6)
3. Run kernel `preScan` on `mpVertsNum` and store the result in `mpVertsBaseAddress`.

`mpVertsBaseAddress`: An array of integers. Its size equals the total number of mesh points. Element `mpVertsBaseAddress[mp]` indicates where vertices, associated with mesh point `mp`, should be stored in the vertex array. It is computed as a prefix-scan of array `mpVertsNum`. So element `mpVertsBaseAddress[mp]` contains the sum of `mpVertsNum` elements up to and excluding `mp`. There is an efficient implementation of prefix-scan algorithm on GPU (see [13]).

4. Allocate memory on GPU for array `verts` of size `vertsNum`.
   `vertsNum`: An integer containing the number of generated vertices.
   It is computed as the sum of the last entries of `mpVertsNum` and its prefix-scanned version `mpVertsBaseAddress`.
   `verts`: An array of floats. Its size equals three times `vertsNum`. It stores the coordinates of the generated vertices. The three coordinates of each vertex are stored consecutively. The vertices associated with each mesh point are stored consecutively from `3*mpVertsBaseAddress[mp]` till `3*mpVertsBaseAddress[mp+1]`. Vertices of a mesh point are ordered according to the local ordering of the edges they lay on (see figure 8).
5. Run kernel `generateVerts` for each mesh point to fill `verts`.
   Vertex coordinates are computed as linear interpolation (see figure 6).

Generating Triangles

1. Allocate necessary memory on GPU for array `mcTriangsNum`.
2. Run kernel `processMC` for each mesh point to fill array `mcTriangsNum`.
3. Run the kernel `preScan` on array `mcTriangsNum` and store the result in array `mcTriangsBaseAddress`.
4. Allocate memory on GPU for array `triangs`.
5. Run kernel `generateTriangs` for each mesh cell to fill array `triangs`.

The semantics are very similar to **Generating Vertices**; Just replace mesh points with mesh cells, edges with tetrahedra and vertices with triangles.
One complication in the final step is getting the addresses of vertices which will be used in forming triangles. Triangles are expressed in terms of the edges on which their vertices lie. So the problem reduces to knowing where the vertex of a certain edge is stored.
If `mp` is lowest numbered mesh point of an edge then the vertex of that edge will be located at `3*mpVertsBaseAddress[mp]` in array `verts` with some additional shift that depends on other vertices associated with `mp`. To get this shift easily, we build an array `mpVertsEdgeIndex` where the bit number `i` of element `mpVertsEdgeIndex[mp]` indicates whether edge number `i` associated with mesh point `mp` is intersected by the isosurface. This way, all the information needed for determining shifts of vertices associated with mesh point `mp` are contained in `mpVertsEdgeIndex[mp]`. The computation of this array is most conveniently done inside kernel `processMP`.

4.4 More Details and Optimization

In the previous description, we omitted several aspects of the method to make the presentation more clear. They are explained here:

- Many subtask like splitting the mesh cell into tetrahedra, getting triangles from tetrahedra pattern, getting shifts from `mpEdgeIndex`, etc. can be done using look-up tables. These look-up tables should be stored in the constant memory of GPU which is basically a small fast read-only memory.
- Considerable speed-up can be obtained by considering only mesh points that actually have a non-zero number of associated vertices and only mesh cells that actually generate triangles. This requires building yet another array for knowing which mesh points are 'active'. Note also that by associating each mesh cell with its lowest numbered mesh point, a mesh cell can be skipped if that mesh point is not active.
- Kernels `processMP` and `processMC` can be combined for speed-up. This because every mesh cell can be associated with its lowest numbered mesh point and then both kernels need to read the same memory locations. Thus, combining them saves repeated memory accesses.
- For getting the normals, central difference formula can be used to get normals at mesh points. This could be done outside the marching tetrahedra algorithm. The normals at isosurface vertices are then computed in the same way as the coordinates .i.e. as a linear interpolation and this computation can be incorporated inside kernel `generateVers`.

4.5 Implementation

Due to time constrains, we developed, as a proof of concept, an application running on CPU but following the GPU-tailored algorithm. An OpenCL application is still under development. The application takes as input GAUSSIAN CUBE format files for reading in the mesh and the scalar data. For full account of complex functions, this file should be accompanied with another file specifying the argument. After loading the data, the user can interactively change the isovalue to see different isosurfaces.

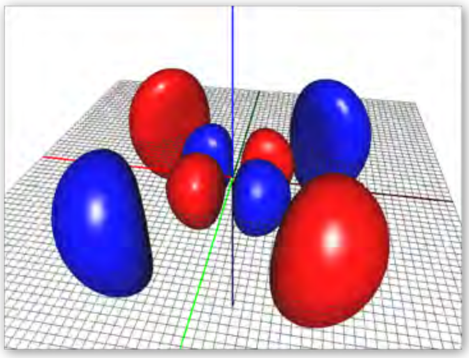
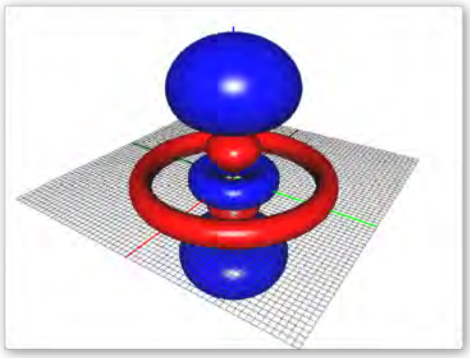

Figure 9: Isosurface plots of hydrogen wave functions $4d_{xy}$(left) and $4d_{3z^2-r^2}$ (right)

5 Summary

We described how to visualize three different classes of complex functions using the GPU. First, we explained how to visualize complex functions of a single complex variable using the domain coloring method on the fragment shader. Then, we explained how to visualize complex function defined on unit sphere by deforming that sphere and coloring it. This is done on the vertex shader. Finally, we explained how to visualize complex functions in 3D space by extracting the isosurfaces of the absolute value using marching tetrahedra method and then coloring that surface. This was designed to work on a GPGPU framework.

6 Acknowledgements

I would like to thank my supervisor Prof. Erik Koch for bringing this interesting program to my attention to and for providing the necessary knowledge and advice to finish the project successfully. I would like also to thank German Research School for Simulation Sciences for sponsoring my project. Last but not least, I would like to thank Jülich Supercomputing Center staff and in particular Mr. Mathias Winkel for organising the program and making it a unique experience.

References

1. Wikipedia contributors. Domain Coloring [Internet]. Wikipedia, The Free Encyclopedia [updated 2012 September 20; cited 2012 Oct 07]. Available from: http://en.wikipedia.org/wiki/Domain_coloring
2. Lundmark M. Visualizing complex analytic functions using domain coloring [internet]. 2004 May [cited 2012 Oct 07]. Available from: www.mai.liu.se/ halun/complex/domain_coloring-unicode.html
3. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical Recipes, The Art of Scientific Computing. 3rd ed. New York: Cambridge University Press;2007. Chapter 6, Special Functions; p.292-295.
4. Lorensen WE, Cline HE. Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH Comput. Graph. 1987 Aug;21(4):163-169.
5. Bourke P. Polygonising a scalar field [Internet]. 1994 May [cited 2012 Oct 04]. Available from: http://paulbourke.net/geometry/polygonise/
6. Doi A, Koide A. An Efficient Method of Triangulating Equivalued Surfaces by using Tetrahedral Cells. IEICE Trans Inf Syst. 1991 Jan;E74(1):214-224.
7. Bourke P. Polygonising a Scalar Field Using Tetrahedrons [Internet]. 1997 Jun [cited 2012 Oct 04]. Available from: http://paulbourke.net/geometry/polygonise/
8. Reck F, Dachsbacher C, Grosso R, Greiner G, Stamminger M. Realtime isosurface extraction with graphics hardware. Eurographics 2004 Short Presentations; 2004.
9. Pascucci V. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Proceedings of IEEE TCVG Symposium on Visualization; 2004. p. 293–300.
10. Klein T, Stegmaier S, Ertl T. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. Proceedings of Pacific Graphics ’04; 2004. p.186–195.
11. NVIDIA CUDA SDK - Physically-Based Simulation - Marching Cubes Isosurfaces [Internet]. 2008 [updated 2009 June 15; cited 2012 Oct 05]. Available from: http://www.nvidia.com/content/cudazone/cuda_sdk/Physically-Based_Simulation.html#marchingCubes
12. NVIDIA OpenCL SDK Code Samples -OpenCL Marching Cubes Isosurfaces [Internet]. [cited 2012 Oct 05]. Available from: http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/website/OpenCL/html/samples.html
13. Harris M, Sengupta S, Owens JD. GPU Gems 3. Addison-Wesley Professional; 2007. Chapter 39. Parallel Prefix Sum (Scan) with CUDA.