Kelvinlets Video Deformation

Guilherme Gomes Haetinger & Eduardo S. L. Gastal

August 3, 2019

New Idea for 3D Interpolation *<2019-05-28 ter>*

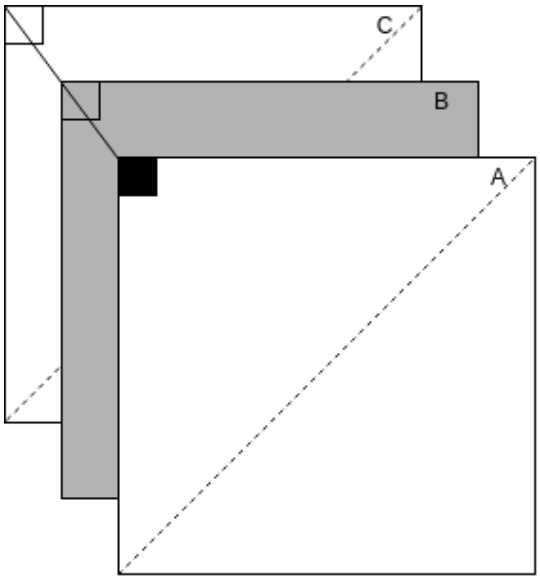

Figure 1: Figure 1: 3D Interpolation with Prism projection. (A) is the first grid involving the frame (B), and (C) is the second one.

Based on the idea of interpolating 3D Cuboids on video frames, I wonder:

are there other (better) ideas for this instead of Interpolating the tetrahedrons inside the prisms? One hypothesis is a prism projection on the video frames [fig.2].

The idea of establishing Scan Conversion on the Frame mesh based solely on line interpolation between the preprocessed grids seems really faster than the tetrahedron approach. Whereas we would process every point inside a bounding box delimited by the corners of every tetrahedron inside a Cuboid (6 volumes), we now process one pixel related to 2 others, which colors will interpolate, and scan convert the remaining pixels of the frame with the produced ones, as we would do in the KelvinletsImageGL code. I like to call this new method 2 step interpolation.

Elaborating the Implementation for the 2 step Interpolation *<2019-05-30 qui>*

Having in mind the Professor's advice to use a bin sort alike algorithm, the main goal is to set which lines collide with each of the frames. We do that by assigning each vertex from the frame having 2 points: one in each deformed frame. Based on that, we can establish the color and position of the vertices. Now, considering the order that the vertices must be processed, we should run them by the time axis, i.e. Interpolate the same pixel position for each frame before moving to another position.

Project Structure *<2019-06-01 sáb>*

Creating the projects structure is a challenge. Separating a OpenGL application in modules is somewhat difficult because of the fact that many functions depend on predeclared variables, which means I need many arguments for a specific function. Therefore, I will, firstly, separate the needed functionalities in main components. These components are:

Main

This module will only serve to process the user arguments set in argv and to call the other methods with these arguments.

VideoDeformer

This module will be in charge of processing the arguments given by the Main modules in order to execute the Kelvinlets grab function in each vertex. It will produce a new set of vertex arrays for the ones in the video input (regularized vertices). This module will include the glm and the glfw3 library, since it will depend on the data structures involving linear algebra as well as the ones defined for OpenGL visualization.

InterpolationFrame

This data structure should hold every line in charge of interpolating each vertex of the frame.

Line

This data structure should hold the information to interpolate a line in a given point in the time axis.

Deformation

Abstraction of a deformation vector(initial position, final position and brush size).

KelvinletsTransformer

Class that provides the execution of the Kelvinlets equation.

VideoVisualizer

This module will be responsible for the video rendering, i.e. the communication of the program with the deformed video data and the graphic card. To do this, we must use the OpenGL library, for which we will need the vertices positions and colors, as well as the order in which they will be accounted for the triangle mesh rendering in the **TRIANGLE STRIP** method. We are supposed to import, for these reasons, **glfw3** and **glew** for the data structures and graphic card communication functions.

For the sake of the project and organization, I also developed a logger for the application. This will make debugging much cleaner and easier. All I have to do to get specific output (errors, fatal errors, warnings and general debugging logs) is call the program in the following way:

```
env KELVIN="FATAL" ./kelvin
#          "ERROR"/"WARNING"/"CORRECT"/"DEBUG"
```

Implementing The Project *<2019-06-04 ter>*

Started with the simpler structures(Deformation, Point and Line). These classes should be working properly, so I will probably test them in the main function before implementing the others, which require more work.

Deformation

Simple getters and setters.

Point

Simple getters.

Line

This one took a little more time than it should. The main functionality for this class was the interpolation of values (color and position). What confused me was the approach that should bee used for this purpose. Firstly, I thought I should use the line between the two points to elaborate its equation and use it to find the xy coordinates for the point that had the Z-axis correspondent to the frame index. That approach was mode complicated, since the 3D line equation is a little bit harder to find and to understand. The best approach ended up being the simple use of a linear interpolation value, as the following equation shows:

$$Z = t * Za + (1 - t) * Zb$$

$$X = t * Xa + (1 - t) * Xb$$

$$Y = t * Ya + (1 - t) * Yb$$

Considering that we already know the value of Z, we can replace its value and find out the following: $t = (ZZa)/(ZbZa)$, which is the reason between the distances.

I ended up having to add more modules for the sake of organization and clean code. These are:

VideoGL

This module is in charge of receiving the input video, setting up its frame's vertex array, color array and index array, initializing the rendering process and calling the RenderableFrames draw method to display the video. It will probably turn the VideoVisualizer obsolete (incorporated more than it was supposed to).

VideoWindow

The idea behind this component is to initialize and instantiate every variable RenderableFrames and VideoGL will need to render into the created window.

Proportions

This is not a module but a header. It was just a way I thought of keeping the width, height and length (the number of frames) altogether.

Implementing the Renderable Video Logic *<2019-07-06 sáb>*

Firstly, I decided to create 2 constructors for VideoGL: One that receives a video path and, thus, will create its own arrays from the video source; and one that will receive a RenderableFrame array and Proportions, which will, than initialize every component needed. This will be needed when 1.reading a video source and 2.turning the InterpolationFrames into renderable.

Initializing the Arrays

Vertex Array

Initializing the vertex array is quite simple: We will need a 2-dimensional for, a variable to keep track of the axis (xy) and another to keep track of the array index. The result of this for will be used as a template for each RenderableFrame. Each one of these will have, now, the value of its z axis built in the class. The needed changes in the project will be necessary to view the video. The code goes like this:

```
int size = this->getWidth() * this->getHeight() * 2;
GLfloat * templateArray = (GLfloat *) malloc(size * sizeof(GLfloat));
int index = 0;
```

```
for(int y = 0; y < this->getHeight(); y++)
  for(int x = 0; x < this->getWidth(); x++){
    templateArray[index] = (GLfloat) normalize(x);
    templateArray[index + 1] = (GLfloat) normalize(y);
    index += 2;
  }
for(int frame = 0; frame < this->getLength(); frame++)
  renderableFrames[frame].setVertices(templateArray);
```

Where normalize takes a coordinate normal to the frame size domain and takes it into the OpenGL screen domain [-1; 1];

Color Array

The basic differences between the creation of the vertex array and the color array is that we will need the video to take the color of its pixels, that there will be no template, since every array will be, probably, different and that the size of the array will be multiplied by 3 (RGB) instead of 2 (xy). To implement the first difference, we will need the Video function to get each color channel, solely. The second difference will be a bit harder, since we will, now, have to create a 3-dimensional for and run through the video frames as we process the new arrays. We can do the second one by something like this:

```
VideoCapture video(videoPath);
//...
int size = this->getWidth() * this->getHeight() * 3;
uint8_t * buff = (uint8_t *)malloc(size * sizeof(uint8_t));
Mat frame;
video >> frame;
memcpy(buff, mat.ptr(0), mat.cols*mat.rows * sizeof(uint8_t) * 3);
//...
//Color for pixel in index
GLfloat R = ((GLfloat) buff[index + 2])/256;
GLfloat G = ((GLfloat) buff[index + 1])/256;
GLfloat B = ((GLfloat) buff[index])/256;
...
```

In this case, index is both the array's and frame's counter. They are both processed the same way.

Index Array

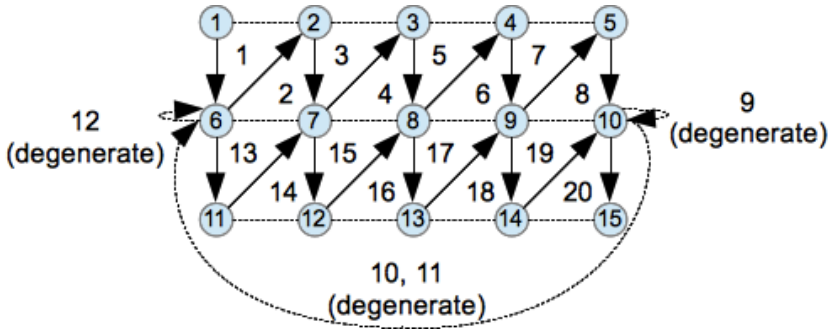

Figure 2: Triangle Strip Scheme

This array, however simple it is to maintain (is always the same or a given proportion), it is the hardest one to generate. Since we will be using the GL TRIANGLE STRIP method of mesh generation for its little memory usage given a regularized vertex grid, we will need to implement the Index array with its rules and restrictions. The best way to do so is to follow the rules as exemplified on fig.2. In every line between the extremes, we should create undrawable triangles (e.g. [6, 6, 2], which has 2 of the same index, which is a line without thickness) so that we can join one side of the triangle with the other, creating a flow. Since there would not be any difference if we apply the redundancy on the extreme lines, we will do that for the sake of simplicity. Processing this array will only happen once, since it is equal in each vertex. The creation function goes like this:

```
int size = getNumberOfIndices(this->getWidth(), this->getHeight());
GLuint * templateArray = (GLuint *) malloc(size * sizeof(GLuint));
int counter = 0;
for(int y = 0; y < this->getHeight(); y++){
  //adds redundancy for 1st element on row
  templateArray[counter] = y * this->getWidth();
  counter++;
  for(int x = 0; x < this->getWidth(); x++){
    //joins the vertex with its correspondent
    //column-wise on the row below
    int columnOutset = y * this->getWidth();
    int position = columnOutset + x;
    templateArray[counter] = position;
```

```
    templateArray[counter+1] = position + this->getWidth();
    counter += 2;
  }
  //adds redundancy for the last element in the row below
  //(before heading to the beginning of the same row)
  templateArray[counter] = (y+2) * this->getWidth() - 1;
}
```

Results for 2 step Interpolation and new Ideas *<2019-07-19 sex>*

Unfortunately, the 2 step Interpolation idea had a strong weakness. Considering that we would project a triangle mesh, some information could be lost by disconsidering some vertices effect on the projection (we considered lines as conductors of the interpolation). For this reason, we had to come up with new and efficient ideas for our problem: displaying each frame with changes in the time axis. Our most promising prospect ideas are drawing each point as a pixel and then use Delunay Triangulation to create rasterizeable triangles, and using any method found for Tetrahedron Slicing.

For the first one, I have to list all pixels for a single frame. To do that, I must create differently sized Buffers for each frame (there can be a great number of pixels in a frame and none in others). Considering that there we are using, for testing, a **640x368x160** sized video, we can estimate that, having $6 * size(GLFloats) + size(GLuint)$ for each one, we will end up using about **1GB** of RAM. We could maybe store only the pixels that have changed frames, which would save a lot of memory, but will be more complex, considering that we would have to search for pixels that have changed to a specific frame every frame drawing iteration, which, in the worst case scenario, would result in a complexity of $O(n^2)$. We could use a hash table, but this is no time for optimization (I think).

For the second one, seen as it is a more abstract method, we will have to search for available implementations or even geometrical equations for Tetrahedron Slicing. This would allow us to get, probably, the same result we had once we implemented the Video Deformation method in Julia, which had a great product but a terrible efficiency. We will be looking into the *CGAL* library and every other material we can find.

CGAL Support for mesh clipping *<2019-07-30 ter>*

Referring to this link: `https://doc.cgal.org/latest/Polygon_mesh_processing/index.html`, we can start testing the function located in the section **3.5**: `CGAL::Polygon_mesh_processing::clip()`. Other options would be the Slicer and the Mesh Intersection. Should start testing the pixel approach and then start a new project to understand these CGAL functions behavior.

Getting this thing to work *<2019-08-02 sex>*

Implementing the idea of rounding each deformed pixel to a frame in the time axis turned out to be a bad solution, for it throws away information. This is shown in the following image:

Figure 3: Figure for the rounded method without corner retardation.

As you can see, there are, clearly, rough edges where there should be a smooth deformation. The triangulation would suffice if we only wanted to fill the black gaps in the video, but that is not our goal (we want to have a smooth image).

To actually do this properly, we will try to use the CGAL library Slicer to clip the tetrahedrons.