Narvik University College

# Fluid Mechanics Visualization
# SMN6200 & STE6234

Pål Solberg Trefall

[a] *Narvik University College, Lodve Langesgt. 2, P.O. Box 385, N-8505 Narvik, Norway.*

11 January 2011

## Abstract

This document is a report of the task, the work, insight and the results from the SMN6200 project in Fluid mechanics visualization at the University College of Narvik, Norway, 2010/2011. Through the project an application was made that would load STL Mesh data and Flow3D simulation data. Using a heavily data-driven approach to the application, most features of the application can be changed and new features can be added, without touching C++ code and without recompiling the software. The scripting interface is based on Lua, which is used for scene initialization and application behaviour. The markup language, XML, has been used to provide a customized way to load any Flow3D data into the application, no matter the set of parameters. The gui uses html/css to define the interface, as is easy to extend with new features. Via the gui, the user can interact with the settings of each single entity loaded into the scene of the application, be it an animated flow3d particle field or a stl mesh. Using the application, the user can analyse the behaviour of particle simulations and follow different kinds of data represented graphically however preferred (the particle field is simply sent to the graphics card, where the geometry shader can be used to customize the visualization of the particles). Bundled with the application there's written a smooth particle visualization, which represents each particle as a spherical point, and a line visualization, which is used to visualize velocity direction and magnitude. The end-result is an application that will render out the particle fields of the assignment, along with the stl meshes, but it's been written with great attention to data-driven architecture and extensibility in mind.

## 1. Introduction

Second half-semester of fall 2010, the project in fluid mechanics visualization was handed out. I immediately envisioned a super-flexible application that would allow a researcher with mild scripting abilities to be able to extend and modify the application to his/her needs. The focus of the application has thus been on a generalized C++ back-end, with a data-driven front-end. This gives the end-user the ability to go through simple text-files to change everything from something as simple to change by which amount you scale an entity when pressing a button, but write entirely new user-interfaces, how geometry/particles are visualized and how logic flows through the application. The user can modify an XML document that describes the parameters to group and expect from his/her flow3d simulations, or can modify the scene start-up script to shoehorn how the application start up.

## 2. Material & methods

### *2.1 Simulation*

The assignment involves a simulation of particles falling down a pipe with a bucket in it, meanwhile water is flowing up from the bottom of the pipe. The simulation shows how all of the particles end up in the bucket due to the energy of the water.

### *2.2 Visualization*

The visualization handles first and most particle field rendering. That is, it renders points on the screen based on a three-dimensional particle resource. How this particle data then is visualized can be heavily modified in the application's data-driven front-end.

Vectors, like velocity, can easily be visualized for a particle-field using the geometry shader in glsl 1.50. Based on the input particle position, we can generate a new vertex-point directly on the Gpu. It would be very easy to extend this to support more info as well. It could generate multiple vectors lit with different colours, it's limitless.

We can also visualize stl triangular meshes. In this application, the face-normals for the mesh is also handled, so that we can perform both lighting calculations, as well as reflect the environment. This can also be extended to perform more advanced visualization techniques if required.

### *2.3 Data-driven application*

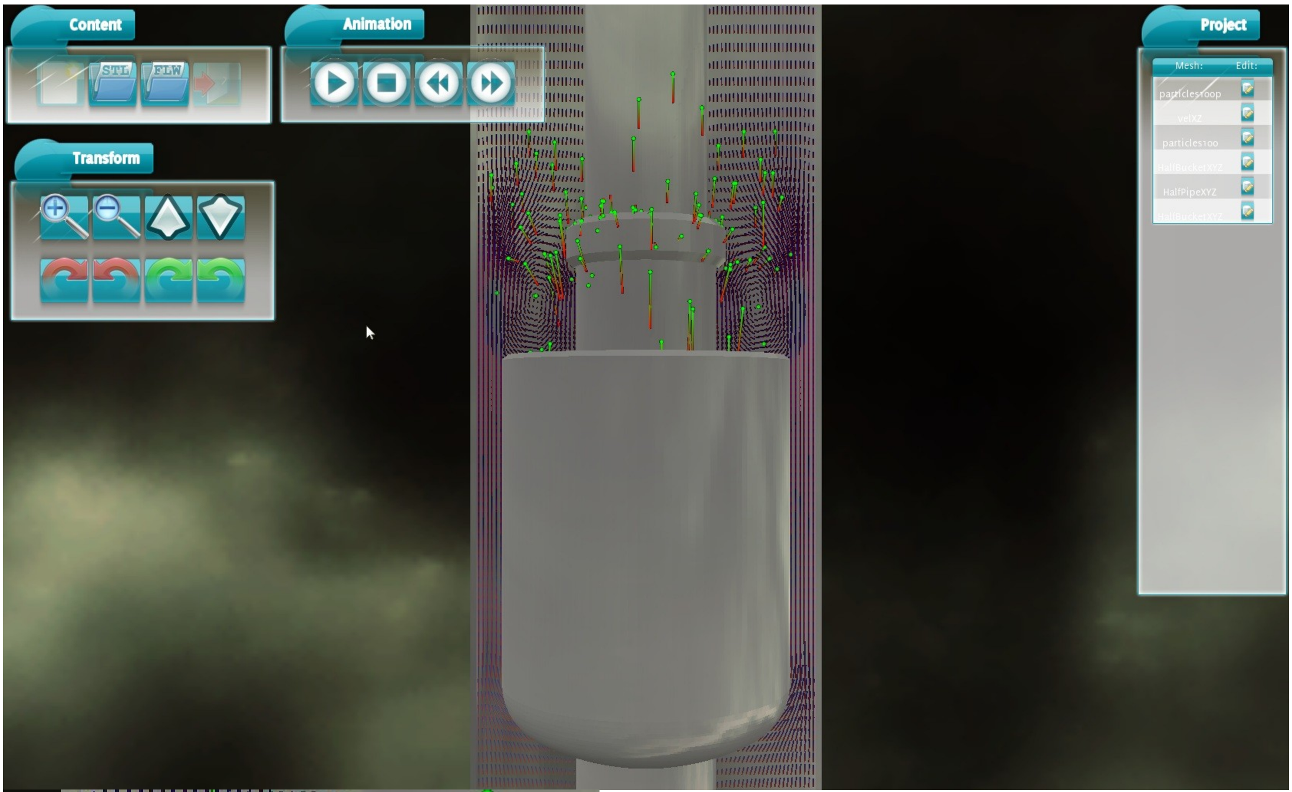
The data-driven nature of the application allows the end-user to change most anything of how the application behaves, and can extend functionality also. Key features in this application's frontend includes:

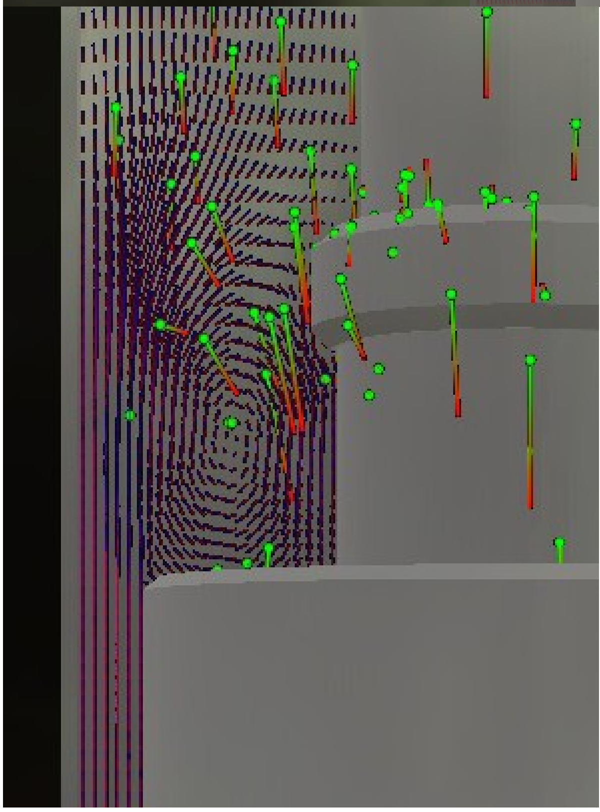
- XML defined parameter grouping for Flow3d file loading
- XML defined configuration of application initialization, allowing the end user to set options like application resolution, pixel depth, fullscreen, vsync, etc.
- XML defined entities, allows the user to define entity types that can later be loaded by the application. Bundled in are skybox, flow3d, stl mesh and cube entities.
- Html/Css-based graphical user interface, allowing the end-user to easily modify and extend the user interface.
  - Can send events straight to Lua scripts.
- Lua scripting, allowing the end-user to do advanced application logic and *talk* with the C++ Engine's exposed features. Add effects via components, change the startup script, tweak behaviour of gui events, and much more!
- GLSL 1.50 scripts, allowing the end-user to alter how particles and other geometry is handled on the graphics-hardware. Visualize points as lines, colour them based on input variables, or whatever the end-user sees fit for the best visual representation to do his/her job.
- Property-system with embedded event system. Allows the end-user to add a property variable of any type to an entity, and add methods that can listen to any existing property when it change state.

## 3. Results

Following is presented some screenshots from the application. An explanation will follow each screenshot, and closeups of particularly interesting sub-sections of each image will be presented too.

### *3.1 Visualization of simulation*



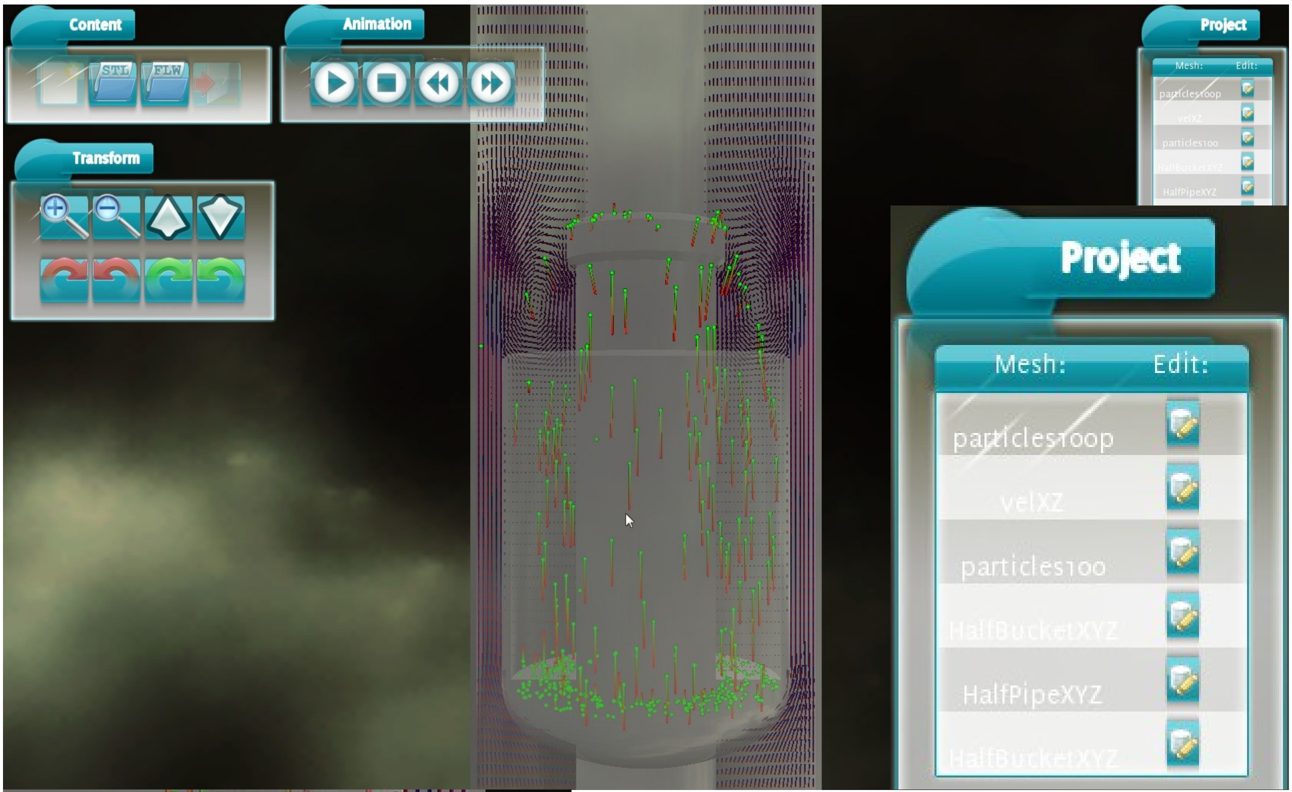
These images shows several feautres of the application's visualization.

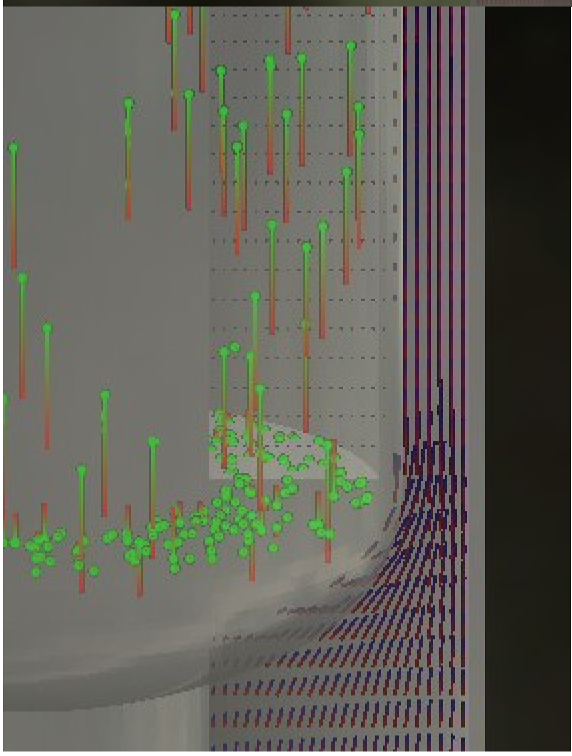
First is the VelocityXZ 2d field data-set that is a one-frame simulation snapshot of the water that clearly flows upward. Blue represents the particle position, and the more red end of the vector represents the velocity magnitude and direction.

It's interesting to see how the particles enters a circular path at the mouth of the bucket, and how fast the particles move on the sides of the bucket, where pressure is very high.

Second is the 100-frame Particle field. It's been loaded in with  a point visualization and a velocity visualization. The green colour represent the particle's position, while the red indicates velocity direction. The longer the vector is, the greater is the magnitude of the velocity.

We can clearly see how particles are forced into the bucket by the force of the water.

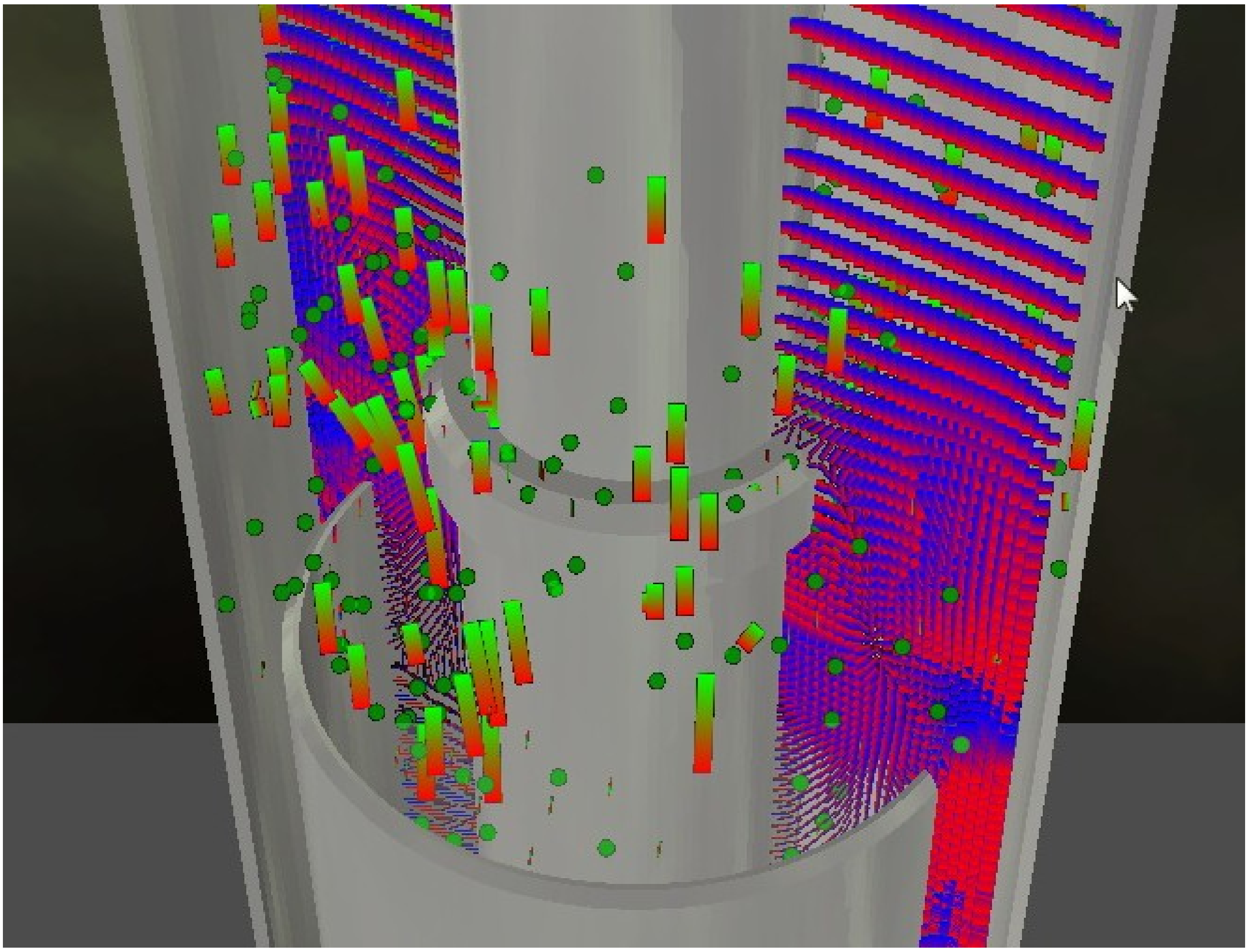


Here the bucket has been set to translucent, effectively allowing us to look inside the bucket. We can see that particles moves towards the bucket floor with high velocity, and bounce against the bucket-floor for a while before they lay to rest.

At the bottom of the bucket, we can also see how the water in the pipe presses into the narrow path between the bucket and the pipe, which results in such a velocity and pressure gain in the water particles.

Zoomed in above, is a snapshot of the Project window on the screen. Every entity loaded into the application is listed here. Each entity has an edit button that can be clicked to edit options for the entity. Here one can alter things like the transparency of the entity, the particle size, the mesh scale, add component-logic modules to the entity, mirror the entity's vertices, and more.



The flexibility of the data-driven application is evident in that it's so easy to change how the scene entities are visualized. Here, some particles are rendered as spheres, others with broad velocity vectors, and the water cells' red velocity tip and blue particle position comes clearly forward. Though this last screenshot might not really provide a lot of useful information, it serves as an example of how simple it is to change the visualization drastically. No code was touched to produce this, and no specialized C++ code was produced either to provide this specific render-state.

### *3.2 Data-driven architecture*

#### *3.2.1 Geometry shader and GLSL 1.50*

```
gl_Position = projMat * mvMat * gl_in[i].gl_Position;
fNormal = gNormal[i];
fViewDir = gViewDir[i];
fLightDir = gLightDir[i];
fParticleIndex = gParticleIndex[i];
fParticleSize = gParticleSize[i];
fColor = vec4(0.0, 1.0, 0.0, 1.0);
EmitVertex();

gl_Position = projMat * mvMat * (gl_in[i].gl_Position + (gVelocity[i]*gScale[i]*0.1));
fNormal = gNormal[i];
fViewDir = gViewDir[i];
fLightDir = gLightDir[i];
fParticleIndex = gParticleIndex[i];
fParticleSize = gParticleSize[i];
fColor = vec4(1.0, 0.0, 0.0, 1.0);
EmitVertex();
```

An excerpt from the geometry shader of the particles that fall into the bucket. First we position the particle and define it's color as green, we then define a second point, which takes the velocity vector as input with a scale modifier, and gives it a red color. Further up in this script, it's clearly stated that we want to produce a line from the input point. The Gpu will automatically interpolate the color value along the line from the first point to the second.

### *3.2.2 Flow3d Parameter groups XML definitions*

```xml
<ParamGroup>
  <Type>ParticlePos</Type>
  <Params>
    <Param>xp</Param>
    <Param>yp</Param>
    <Param>zp</Param>
  </Params>
</ParamGroup>

<ParamGroup>
  <Type>ParticleVel</Type>
  <Params>
    <Param>up</Param>
    <Param>vp</Param>
    <Param>wp</Param>
  </Params>
</ParamGroup>

<ParamGroup>
  <Type>ParticleSize</Type>
  <Params>
    <Param>psiz</Param>
  </Params>
</ParamGroup>

<ParamGroup>
  <Type>ParticleIndex</Type>
  <Params>
    <Param>index</Param>
  </Params>
</ParamGroup>
```

### *3.2.3 Html/Css-based gui formating and definition*

```
<form onsubmit="FLWOptionsWindow:OnSubmitClicked; close">
  <div>
    <p>
      Alpha Value:
      <select name="alphaselect" id="alphaselect">
        <option name="opaque" value="opaque">Opaque</option>
        <option name="transparent" value="transparent">Transparent</option>
        <option name="hide" value="hide">Hide</option>
      </select>
    </p>
    <p>
      Point Size:
      <select name="pspread_select" id="pspread_select">
        <option name="tiny" value="tiny">Tiny</option>
        <option name="small" value="small">Small</option>
        <option name="medium" value="medium">Medium</option>
        <option name="large" value="large">Large</option>
      </select>
    </p>
    <p>
      Show Half: <input type="checkbox" name="half_check" value="half"/>
    </p>
    <p>
      Add Component:
      <input id="add_comp" type="text" name="add_comp" value="" />
    </p>
  </div>
  <input style="width: 40px; height: 40px;" type="submit" value="ok">
    <img src="btn_ok2.tga" />
  </input>
  <input style="width: 40px; height: 40px;" type="submit" value="close">
    <img src="btn_no2.tga" />
  </input>
</form>
```

### *3.2.4 Lua scripted gui event handling and property-system usage*

```
function FLWOptionsWindow:OnSubmitClicked(event)
    local submit = event.Parameters["submit"]
    if(submit == "ok") then
        local entity = GetSelectedEntity()
        if(entity == nil) then
            return
        end

        local alpha = event.Parameters["alphaselect"]
        if(alpha ~= nil) then
            if(alpha == "opaque") then
                entity:SetAlpha(1.0)
            elseif(alpha == "transparent") then
                entity:SetAlpha(0.5)
            elseif(alpha == "hide") then
                entity:SetAlpha(0.0)
            end
        end

        local half = event.Parameters["half_check"]
        if(half ~= nil and half ~= "") then
            entity:SetShowInHalf(true)
        else
            entity:SetShowInHalf(false)
        end
```

### *3.2.4 Lua scripted logic component and more property demonstration*

```

if(SpinEffect == nil) then
    SpinEffect = {}
end

function SpinEffect:OnInit(entity)
    entity:AddProperty("Yaw", 0.0)
    entity:AddProperty("YawRate", 1.0)
end

function SpinEffect:OnUpdate(entity, dt)
    entity:SetYaw(entity:GetYaw()+entity:GetYawRate()*dt*0.2)
end

RegisterComponent("SpinEffect")

```

## 4. Conclusions

Though I never quite reached all the goals that I set out with, I felt that the most important parts I wanted to focus on with the project was completed quite well.

I started a binary project format, that was meant to store the current scene with everything from camera placement, to all the entities that were loaded into the scene, and their settings. The property system on entities was perfect for serialization support, and the basic structure to make this possible has all been written and integrated into the project. I never got time to complete the save/load part of this file-format however.

I also implemented a worker-thread pool. This pool of threads spawns worker threads to the amount of cores on the computer that's running the application. The idea was to use this API to load in files from the hard-drive without freezing the application, and other heavy calculations that could be handled in a background thread. I never found time to actually use this API.

Writing a generalized loader for Flow3d ascii files proved to be quite challenging, but I think my solution ended up quite flexible. It does require, however, that every new simulation file also gets it's own GLSL shader. This rule is bound in script, however, and could easily be modified by an end-user. It's quite convenient, however, to be able to modify these shaders for each specific simulation.

I focused heavily on data-driven architecture and a good, intuitive user interface. The html/css interface to gui elements made it very easy to work with, but there was no built-in support for here for convenient features, like browsing files on disk. The application ended up very flexible however. It can indeed be heavily modified and extended without touching a single line of C++ code!

I felt I ran a bit short on time for focusing more on the simulation itself. I was too busy making a generalized Flow3d visualization application. I did get in quite a few advanced features in the end, like the velocity vectors for particles, but I had a lot more planned.

I really wanted to make available a general graph window that the end-user could insert specific information into and track it, perhaps with the possibility to support advanced, specialized plotting via Lua scripts. It wouldn't take much to add in such a feature.

I also really wanted to make available some kind of collision-field. The collision-field could be a simple plane, or a more sophisticated geometric entity (maybe even an stl mesh) that the user could place in the scene, and it would send events to the Lua scripting-layer when collisions occurred. This feature wasn't too far fetched to get in with a couple days more time either.

In the end, I got in enough features to give the end-user some tools to analyse the simulation data. Perhaps more importantly though, the application has been written in such a way that extensibility and improvement of the application would be easy. This application wasn't thrown together, but carefully designed to provide a flexible and extensible tool-set that could be easily modified by an end-user.

## Acknowledgements

The author would like to thank his fellow students for sharing know-how and ideas on this complex project. Also a thanks is given to the developers behind ClanLib GDK, libRocket and LuaPlus; -this project would never have reached this level without those embedded APIs.

## References

Flow Science Inc., 2010, 683 Harkle Rd Ste A, Santa Fe, NM 87505, http://www.flow3d.com/apps/index.html (21.10.2010).

ClanLib GDK, 2010, Game-Development-Kit, http://clanlib.org  (12.01.2011).

libRocket GUI, 2010, Html/Css Game Interface Middleware, http://librocket.com/ , (12.01.2011).

Trefall, P.S., 2009, Component-based entity system using properties, http://www.gamedev.net/topic/551515-component-based-entities-using-properties/ , (12.01.2011).