Title-Needed: A framework for interactive computational fluid simulation and visualization

September 5, 2017

1 Introduction

This is a framework for interactive simulation and visualization of steered computational fluid and rigid body simulations. It is based on pipelining concept further explained in Sec. **??** and a modular communication infrastructure.

This document gives an introduction to the basic concepts of the framework and describes the requirements of it.

2 Requirements & Compilation

The project was developed to be compiled and executed on Ubuntu 16.04 LTS [1].

After downloading and installing the system, several packages are required. Those can be installed using *apt-get* from the command line.

```
$ apt-get install libsdl2-dev libsdl2-image-dev g++
$ apt-get install git
```

For eclipse development environment, additionally install java jre

```
$ apt-get install default-jre
```

After installation of the required packages, the code can be compiled with

```
$ make
```

In case that you get any compilation errors, please send your advisors an email. The program will be located in the build folder and can be run by

```
$ ./build/fa_2017_release
```

[1] `http://www.ubuntu.com/download/desktop`

3 Classes

There are a bunch of existing classes which should be used by everyone to setup the interfaces among the groups. The following table gives an overview and short description of each class.

CDataArray2D.hpp	Data storage for 2D arrays
CDataDrawingInformation.hpp	Data storage for interactive drawing information
CGlTexture.hpp	Abstraction for OpenGL Textures
CParameters.hpp	Program and simulation parameters
CPipelinePacket.hpp	Pipeline packet capable of being forwarded via the pipeline
CPipelineStage.hpp	Pipeline stage providing interfaces for pipelining
CSDLInterface.hpp	SDL Interface for visualization and interactivity
CStage_ImageInput.hpp	Pipeline stage for image input (single image from file)
CStage_ImageProcessing.hpp	Pipeline stage for image processing filter
CStage_VideoInput.hpp	Pipeline stage for video input, e.g. from webcam
CStage_VideoOutput.hpp	Pipeline stage for video output
main.cpp	main entry to setup pipelined scenarios

4 Pipeline

The idea of a pipelining model is creating independent execution parts with particular input-output specifications. E.g. a webcam only provides images which are forwarded to the image filter. After processing of the image filter, this information is further forwarded to the simulation (not yet implemented) and then to the output for visualization.

All those pipeline stages are independent and thus can be independently processed - e.g. Image filter and simulation computations in parallel.

4.1 Pipeline stage

We continue with an example given by the image processing filter *CStage_ImageProcessing.hpp*. For well-known interfaces, each new pipeline stage has to inherit the class CPipelineStage:

```
1 class CStage_ImageProcessing    :       public
2         CPipelineStage
3 ...
```

For processing of the images, paramters are required to know which computations to do, at least a single input storage is required as well as an output

storage to forward processes images to other classes:

```
...
/**
 * global parameters
 */
CParameters &cParameters;

/**
 * input image
 */
CDataArray2D<unsigned char,3> input_cDataArray2D;

/**
 * processed image
 */
CDataArray2D<unsigned char,3> output_cDataArray2D;
...
```

Since the parameters are shared with the other classes, they are setup in the constructor:

```
public:
/**
 * constructor
 */
CStage_ImageProcessing(CParameters &i_cParameters):
CPipelineStage("ImageProcessing"),
cParameters(i_cParameters)
{
}
```

In case of an input sent via the pipeline of another stage such as the video input, the method *pipeline_process_input* is executed and has to be implemented. This interface is particularly requested by the class *CPipelineStage*.

```
void pipeline_process_input(
        CPipelinePacket &i_cPipelinePacket
)
{
...
```

Since not all possible data types can be probably processed, we have to check for compatible input packages and unpack the data to make it available with our accessor class:

```
// we are currently only able to process "unsigned char
   ,3" data arrays.
if (i_cPipelinePacket.type_info_name != typeid(
   CDataArray2D<unsigned char,3>).name())
{
        std::cerr << "ERROR: Video Output is only able to
            process (char,3) arrays" << std::endl;
        exit(-1);
}

// unpack data
```

```
CDataArray2D<unsigned char,3> *input = i_cPipelinePacket.
    getPayload<CDataArray2D<unsigned char,3> >();
```

After unpacking the data and processing the data with more details available in the source code file itself, the output data array is pushed to the pipeline and thus forwarded to the next pipeline stages:

```
CPipelineStage::pipeline_push((CPipelinePacket&)
    output_cDataArray2D);
```

4.2 Pipeline setup

After programming several stages, their input and output has to be connected after instantiation. E.g. let us assume that we like to have an static input image with an image filter and the possibility to draw into the image, this would lead to the following pipeline:

```
// static image input
CStage_ImageInput cStage_ImageInput(cParameters);
// video output
CStage_VideoOutput cStage_VideoOutput(cParameters);

// PIPELINE CONNECTIONS
// forward image to video output
cStage_ImageInput.connectOutput(cStage_VideoOutput);
// forward mouse movements to image input
cStage_VideoOutput.connectOutput(cStage_ImageInput);


// initial push of static image
cStage_ImageInput.pipeline_push();

// main loop
while (!cParameters.exit)
{
        // trigger image input to do something
        cStage_VideoOutput.main_loop_callback();
}
```

For our pipeline concept, only the outputs have to be connected. The initial push for the image input is required to initially forward the static image to the video output.

The main loop is required to e.g. check for user input, to draw updates for the visualization and to run a simulation timestep.

5 Interaction

Several keystrokes currently exist updating some parameters in the class *CParameters*. Note that all pipeline stages get a reference to this class during initialization.

Since the Video output is closely connected to the input system, all input keystrokes which are not directly processed are forwarded to the method key_down of the parameter class:

```
/**
 * return bool if processed
 */
bool key_down(char i_key)
{
        switch(i_key)
        {
        case SDLK_j:
                stage_imageprocessing_filter_id--;
                std::cout << "Using filter id " <<
                    stage_imageprocessing_filter_id <<
                    std::endl;
                return true;

        case SDLK_k:
                stage_imageprocessing_filter_id++;
                std::cout << "Using filter id " <<
                    stage_imageprocessing_filter_id <<
                    std::endl;
                return true;
```

This allows the modification of the parameters during the programs runtime. So far the following keystrokes are defined:

General	
q	quit program

1: image Processing	
j,k	decrease / increase filter id
g,t	decrease/increase threshold value

6 Programn start

Several program parameters currently exist and are also processed in CParameters:

General	
p	pipeline id to use (see main.cpp)
v	verbosity level
0: image/videoinput	
d	video device string to use
w	request this width for video input
h	request this height for video input
i	path to input image to use
3: fluid simulation LBM	
v	switch between flag field and velocity output
4: parallelization	
n	number of threads to use

7 Group 0: Management

8 Group 1: Input devices

More information for installation of the Kinect drivers for linux is available in kinect.txt

9 Group 2: Image processing

10 Group 3a: Lattice Boltzmann simulation

11 Group 3b: Navier-Stokes simulation

12 Group 4: Parallelization

13 Group 5: Rigid body simulation

14 Group 6: Visualization