Frame Buffers

Andrea Tagliasacchi

Changes to CMake

```
get_filename_component(EXERCISENAME ${CMAKE_CURRENT_LIST_DIR} NAME)

file(GLOB_RECURSE SOURCES "*.cpp")
file(GLOB_RECURSE HEADERS "*.h")
file(GLOB_RECURSE SHADERS "*.glsl")

add_executable(${EXERCISENAME} ${SOURCES} ${HEADERS} ${SHADERS})
target_link_libraries(${EXERCISENAME} ${COMMON_LIBS})
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_CURRENT_LIST_DIR})
```

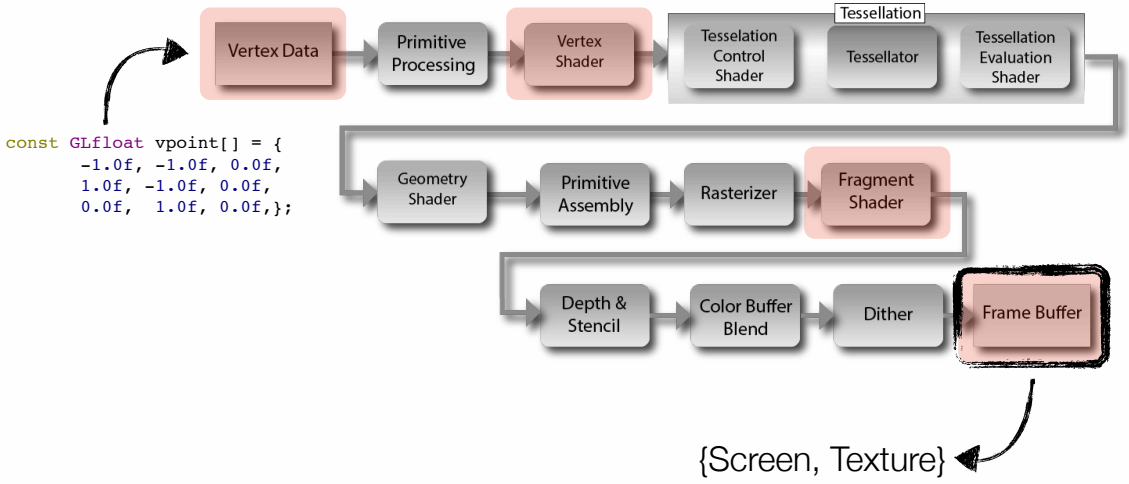
to have shaders "visible" in your IDEs

executable deployed in the source folder (no need to copy shaders)

```
void Cube::init(){
    ///--- Compile the shaders
    _pid = opengp::load_shaders("_cube/cube_vshader.glsl",
                                "_cube/cube_fshader.glsl");
    if(!_pid) exit(EXIT_FAILURE);
    glUseProgram(_pid);
    ///...
}
```

remember to access **shaders** and **textures** with the right path now!!!

OpenGL Pipeline



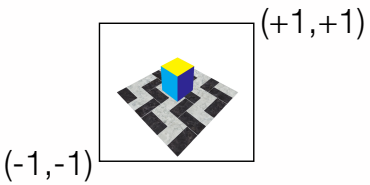

```
const GLfloat vpoint[] = {
       -1.0f, -1.0f, 0.0f,
       1.0f, -1.0f, 0.0f,
       0.0f,  1.0f, 0.0f,};
```


Screen is the “default framebuffer” (with ID=0) and OpenGL creates it automatically for you

Using the FrameBuffer

Pass #1
render scene to a texture/s

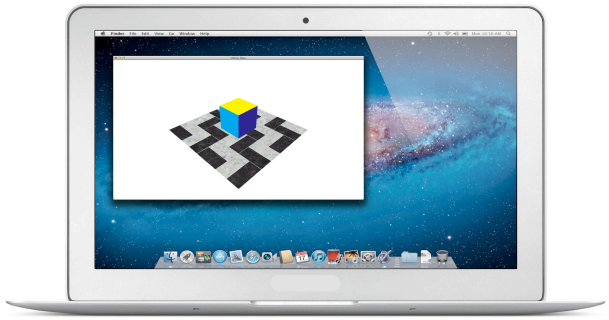
```
static const vec3 cubeVertices[] = {
    vec3(-0.5, -0.5, -0.5),
    vec3(-0.5, 0.5, -0.5),
    vec3(0.5, -0.5, -0.5),
    ///...
```




```
out vec3 color; ///< texture!
```

Pass #2
render texture with a fullscreen quad

```
uniform sampler2D tex;
void main(){ color = texture(tex,uv).rgb; }
```



Using the FrameBuffer

```
/// @file framebuffer/main.cpp
#include "FrameBuffer.h"
///...
FrameBuffer fb(width, height);
ScreenQuad squad;
///...

void init(){
    ///...
    GLuint fb_tex = fb.init();
    squad.init(fb_tex);
}

void display(){
    ///...

    ///--- Render to FB
    fb.bind();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        cube.draw(VP, glfwGetTime());
        quad.draw(VP);
    fb.unbind();

    ///--- Render to Window
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    squad.draw();
}
```

declaration/initialization

anything you draw after the framebuffer is bound will be saved in the texture *fb_tex*

Finally we render the *fb_tex* by drawing a simple (textured) quad **(see previous practicals!!!)**

What you see above will go in from framebuffer/main.cpp
We provide FrameBuffer.h as part of the framework. Students struggled *a lot* last year to setup this correctly (very difficult to debug a white screen).

Using the FrameBuffer

```
/// @file framebuffer/main.cpp
#include "FrameBuffer.h"
///...
FrameBuffer fb(width, height);
ScreenQuad squad;
///...

void init(){
    ///...
    GLuint fb_tex = fb.init();
    squad.init(fb_tex);
}

void display(){
    ///...

    ///--- Render to FB
    fb.bind();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        cube.draw(VP, glfwGetTime());
        quad.draw(VP);
    fb.unbind();

    ///--- Render to Window
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    squad.draw();
}
```

Using the FrameBuffer

How much of that buffer should we re-draw?

where the next glDraw*(...) will be drawing?

```
void Framebuffer::bind() {
    glViewport(0, 0, _width, _height);
    glBindFramebuffer(GL_FRAMEBUFFER, _fbo);
    const GLenum buffers[] = { GL_COLOR_ATTACHMENT0 };
    glDrawBuffers(1 /*#elems in buffers[]*/, buffers);
}

void Framebuffer::unbind() {
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

After we are done, we bind the default framebuffer (screen)

Framebuffers have multiple attachments. Which one do we want to use?
our _color_tex will be set to #0

We will see how to create the texture and the framebuffer in a few slides.
If you have more than one attachment (you have more than one "out" in the fragment shader) you have to modify the buffers[] array to include them:

```
const GLenum buffers[] = { GL_COLOR_ATTACHMENT0,
GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2 /*#elems in buffers[]*/, buffers);
```

Multiple Attachments

If you have multiple attachments, remember to **clear** them!
(see the homework code for the motion_blur)

```
void display() {
    //...
    fb.clear();
    fb.bind();
        _quad.draw(MVP);
    fb.unbind();
    //...
}
```

```
void Framebuffer::clear(){
    glViewport(0, 0, _width, _height);
    glBindFramebuffer(GL_FRAMEBUFFER, _fbo);
        glDrawBuffer(GL_COLOR_ATTACHMENT0);
        glClearColor(1.0, 1.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glDrawBuffer(GL_COLOR_ATTACHMENT1);
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```


The framebuffer we used in the practical was simple (one attachment), so clearing it was identical to traditional OpenGL. Unfortunately glClear is not smart enough to work on multiple attachments, i.e. glDrawBuffers(...)

Using the FrameBuffer

```
/// @file framebuffer/main.cpp
#include "FrameBuffer.h"
///...
FrameBuffer fb(width, height);
ScreenQuad squad;
///...

void init(){
    ///...
    GLuint fb_tex = fb.init();
    squad.init(fb_tex);
}

void display(){
    ///...

    ///--- Render to FB
    fb.bind();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        cube.draw(VP, glfwGetTime());
        quad.draw(VP);
    fb.unbind();

    ///--- Render to Window
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    squad.draw();
}
```

Framebuffer Setup 1/3

```
int Framebuffer::init(bool use_interpolation = false) {
    ///--- PART 1/3: Create color attachment
    {
        glGenTextures(1, &_color_tex);
        glBindTexture(GL_TEXTURE_2D, _color_tex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

        if(use_interpolation){
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        } else {
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        }

        ///--- Create texture for the color attachment
        glTexImage2D(GL_TEXTURE_2D, 0 /*level*/, GL_RGB8, _width, _height, 0 /*border*/,
                     GL_RGB, GL_UNSIGNED_BYTE, NULL); ///< how to load from buffer
    }

    ///--- PART 2/3: Create render buffer ...
    ///--- PART 3/3: Tie it all together ...

    return _color_tex;
}
```

RGB texture as large as the whole screen

only necessary when the last argument is not NULL (i.e. load tex from memory)

What you see above comes from framebuffer/Framebuffer.h

Framebuffer Setup 2/3

```
int Framebuffer::init(bool use_interpolation = false) {
    ///--- PART 1/3: Create color attachment ...

    ///--- Part 2/3: Create render buffer (for depth channel)
    {
        glGenRenderbuffersEXT(1, &_depth_rb);
        glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, _depth_rb);
        glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT32, _width, _height);
        glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);
    }

    ///--- PART 3/3: Tie it all together ...

    return _color_tex;
}
```

we want to work with 3D geometry - we need depth-test
think of “RenderBuffers” as a texture to store depth

What you see above comes from framebuffer/Framebuffer.h
We need to create this depth buffer manually, as manually created framebuffers don't have it!!!

Framebuffer Setup 3/3

```
int Framebuffer::init(bool use_interpolation = false) {
    ///--- Part 1/3: Create color attachment ...
    ///--- Part 2/3: Create render buffer (for depth channel)

    ///--- Part 3/3: Tie it all together
    {
        glGenFramebuffers(1, &_fbo);
        glBindFramebuffer(GL_FRAMEBUFFER_EXT, _fbo);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 /*loc=0*/, GL_TEXTURE_2D, _color_tex, 0/*level*/);

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, _depth_rb);

        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
            std::cerr << "!!!ERROR: Framebuffer not OK :(" << std::endl;
        glBindFramebuffer(GL_FRAMEBUFFER, 0); ///< avoid pollution
    }

    return _color_tex;
}
```

create + bind framebuffer

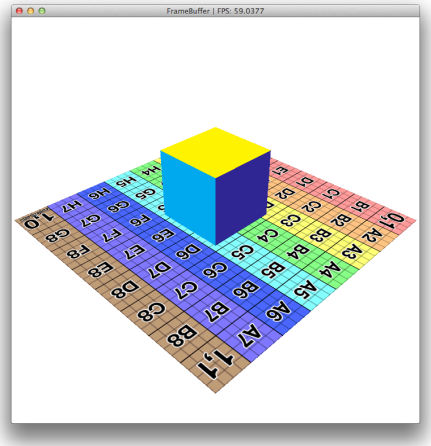
```
/// fshader: tells which texture to write
layout (location = 0) out vec3 color;
```

check everything is “ok”

as we need depth test, tell the FrameBuffer where to perform depth buffer operations

earlier we just created textures (depth+color), now we need to tell them how these are assembled to crate a full framebuffer!
Note the layout specifier was automatically set for you in earlier exercises. If you don't specify it "out" variables will be automatically enumerated location={0,1,2,3,...}.
When (homeworks) you are working with more than one target I *highly* reccomend you to manually specify it: e.g. "layout (location=0) out vec4 color;"

TASK: “passthrough” FB



Read the provided code and **ask questions!!!**
Given: cube+plane scene rendered
TASK #1: render scene to texture (using framebuffer)
TASK #2: render scene to fullscreen quad

(i.e. copy paste from these slides)

NOTE: you can control the cube’s animation by commenting/uncommenting one line in its vertex shader
NOTE: for this task you only need to modify the content of main.cpp

Convolution

convolution

$$I_{new} = G_x * I = ?$$

careful with indexes!!
(0,0) is the filter center here

$$I_{new}(x,y) = \sum_{j=-J}^{j=+J} \sum_{k=-K}^{k=+K} G_x(j-x, k-y) I(x,y)$$

$$I = \begin{bmatrix} I_{00} & I_{01} & I_{02} & I_{03} \\ I_{10} & I_{11} & I_{12} & I_{13} \\ I_{20} & I_{21} & I_{22} & I_{23} \\ I_{30} & I_{31} & I_{32} & I_{33} \end{bmatrix}$$

$$I_{new}(1,1) = ?$$

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

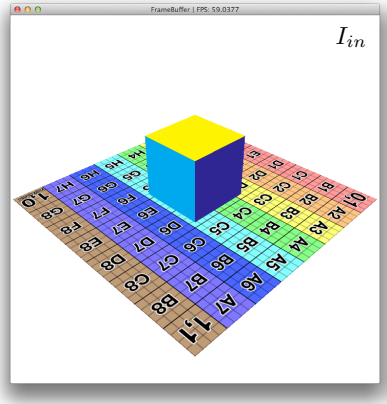
j=0

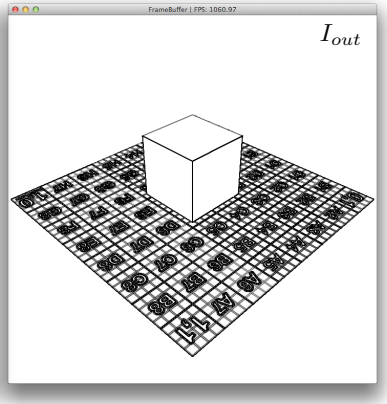
k=0

$$\begin{aligned} I_{new}(1,1) &= -\mathbf{1}\, I(0,0) + \mathbf{0}\, I(0,1) + \mathbf{1}\, I(0,2) \\ &= -\mathbf{2}\, I(1,0) + \mathbf{0}\, I(1,1) + \mathbf{2}\, I(1,2) \\ &= -\mathbf{1}\, I(2,0) + \mathbf{0}\, I(2,1) + \mathbf{1}\, I(2,2) \end{aligned}$$

SEE: http://en.wikipedia.org/wiki/Convolution#Discrete_convolution
The reason for which I use the (0,0) convention above is that G might be a close form function (a 2D gaussian for example)

TASK: edge detection





$$I_{out} = 1.0 - \sqrt{(G_x * I_{in})^2 + (G_y * I_{in})^2}$$

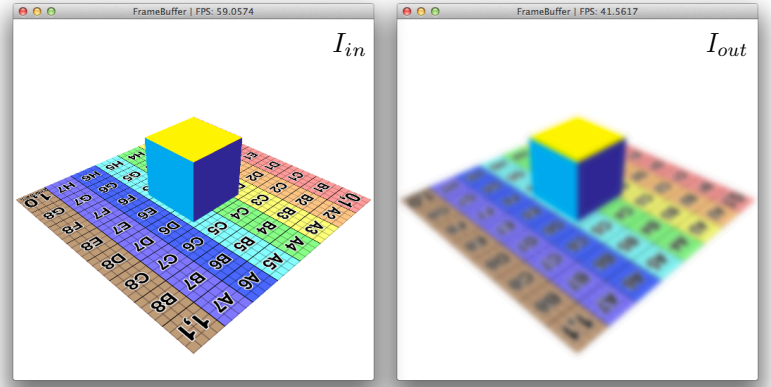
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

relative position w.r.t. current fragment

```
float t_01 = -2 * rgb_2_luma( textureOffset(tex, uv, ivec2(-1,  0)).rgb );
```


NOTE: only necessary to modify the **ScreenQuad_fshader.glsl**

TASK: gaussian blur



$$I_{out} = I_{in} * \mathcal{N}$$

$$\mathcal{N}(x) = e^{-\frac{x^2}{2\sigma^2}}$$

$$\sigma = 2$$

N = fspecial('gaussian', 11, 2); /// MATLAB

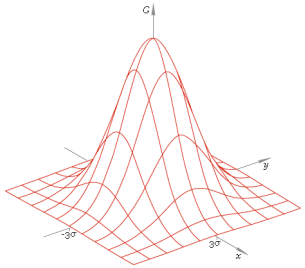
N = 0.0001	0.0002	0.0006	0.0011	0.0016	0.0018	0.0016	0.0011	0.0006	0.0002	0.0001
0.0002	0.0007	0.0018	0.0033	0.0048	0.0054	0.0048	0.0033	0.0018	0.0007	0.0002
0.0006	0.0018	0.0042	0.0079	0.0115	0.0131	0.0115	0.0079	0.0042	0.0018	0.0006
0.0011	0.0033	0.0079	0.0148	0.0215	0.0244	0.0215	0.0148	0.0079	0.0033	0.0011
0.0016	0.0048	0.0115	0.0215	0.0313	0.0355	0.0313	0.0215	0.0115	0.0048	0.0016
0.0018	0.0054	0.0131	0.0244	0.0355	0.0402	0.0355	0.0244	0.0131	0.0054	0.0018
0.0016	0.0048	0.0115	0.0215	0.0313	0.0355	0.0313	0.0215	0.0115	0.0048	0.0016
0.0011	0.0033	0.0079	0.0148	0.0215	0.0244	0.0215	0.0148	0.0079	0.0033	0.0011
0.0006	0.0018	0.0042	0.0079	0.0115	0.0131	0.0115	0.0079	0.0042	0.0018	0.0006
0.0002	0.0007	0.0018	0.0033	0.0048	0.0054	0.0048	0.0033	0.0018	0.0007	0.0002
0.0001	0.0002	0.0006	0.0011	0.0016	0.0018	0.0016	0.0011	0.0006	0.0002	0.0001

i=-4, j=-2

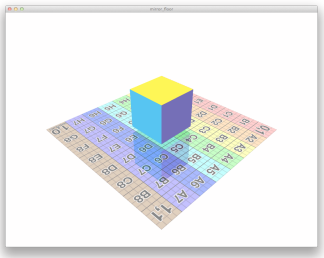
```
uniform float tex_width;
uniform float tex_height;
texture(tex, uv+vec2(i/tex_width,j/tex_height)).rgb;
```


NOTE: only necessary to modify the ScreenQuad_fshader.glsl
NOTE: use a (nested) for loop in the fragment shader to compute the convolution of RGB pixels with a gaussian kernel.
NOTE: you **cannot use textureOffset**, as the parameters of texture shift are not constants (shader compiler error)
NOTE: textureOffset works in “pixel” space [0...WindowWidth] conversely texture(...) works in UV coordinates [0...1].
NOTE: if you do this the “dumb” way by manually entering the coefficients, N is already normalized sum(N)=1, **when you do it with the for look remember to normalize by the sum of coefficients!**
NOTE: sigma=2, therefore stddev~=1.4, gaussian needs 3xstddev to reach approx zero, so you need a stencil of size 4.5. But this has to be on both sides, so we need a **stencil (i.e. for loop) of at least size 9**!

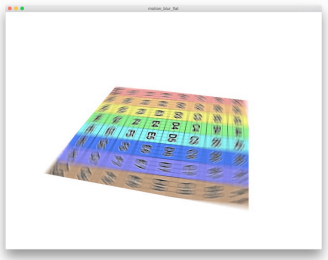
Homework 4



Fast Gaussian Blur
(2 points)



Floor Mirror Effect
(2 points)



Motion Blur
(2 points)

Open Ended Bonus
(2 points)

(maximum grade 6/6)

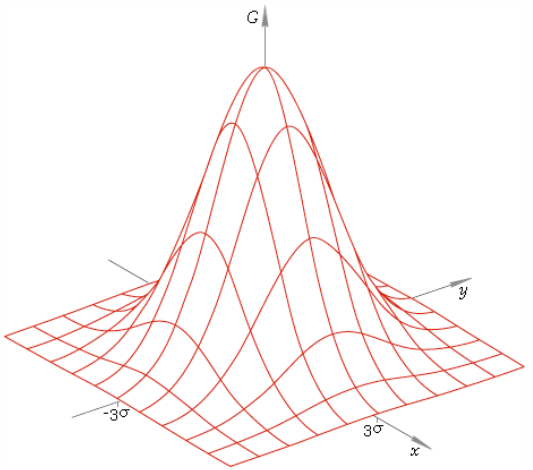
Open Ended: is there something regarding framebuffers you'd like to experiment with? Talk with the TAs to get it "approved".

Efficient Gaussian Filtering

Gaussian filters are separable in {x,y}, we can compute convolution with a two pass filter!

$$I_{out} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} * I_{input} = \frac{1}{4}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix} * \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} * I_{input}$$

$$I_{tmp} = G_x * I_{input}$$
$$I_{out} = G_y * I_{tmp}$$

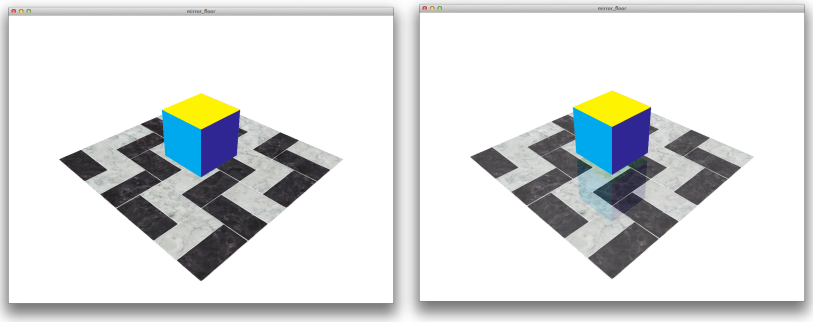


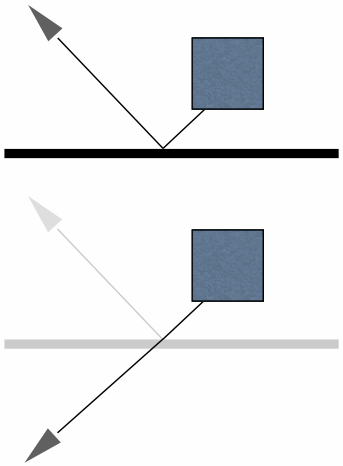
*) compute **1D kernel G** in CPU and pass it to shader as **float[]**
*) modify the framebuffer to have **2 attachments/textures**
*) link keyboard **Q/W** changes variance of filter by **+/-0.25**

Implement the following 3-pass rendering:
1) apply the **convolution along x** and save to *tmp* texture
2) apply the **convolution along y** on *tmp* texture (save on tmp2)
3) display the fullscreen **quad (with the *tmp2* texture)**

TODO: bind Key_1, Key_2 to swap between brute force (from practical) and optimized (from homework) versions of gaussian filtering (expect to observe large difference in FPS!!!)
TODO: bind Key_Q and Key_W as you did in the practical to change the filter variance (you need to adapt the float[] stencil size dynamically)
TODO: you will need to create two textures (as you cannot simultaneously read and write the results of a filtering operation). One will be bound to input, the other to output.
NOTE: the 3x3 kernel above is JUST an example to show you the decomposition in two filtering steps
NOTE: two pass reduces the complexity of filtering one pixel from N^2 to N.
SEE: http://en.wikipedia.org/wiki/Separable_filter

Screen Space Reflections





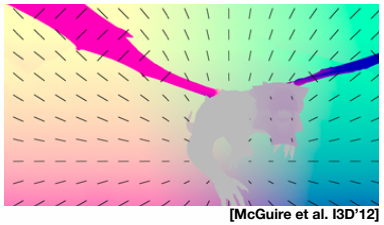
Realize the realtime mirroring as a 2-pass rendering:
1) mirror camera w.r.t. floor and render the cube to a FrameBuffer
2) render the scene consisting of cube **and** floor*
3) ...but modify floor shader to **blend** the tiled texture with the mirrored!

```
color = mix(color_from_texture, color_from_mirror, vec3(.15)); ///< composite texture with mirror image
```


NOTE: you will have to flip the texture from the camera upside down
NOTE: use the predefined gl_FragCoord to access the mirror texture!

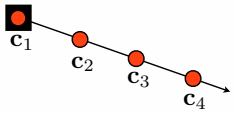
Screen Space Motion Blur

Per-Pixel Motion Vectors



[McGuire et al. I3D'12]

Directional convolution (with box filter)



$$\mathbf{c}_{new} = \frac{1}{N}\sum_{1}^{N}\mathbf{c}_i$$

Compute the motion blur filtering as a **two pass** rendering:
1) Compute a velocity vector (vec2: in pixels) and save it in an FB attachment
2) In the final pass perform a **directional convolution** using the velocity vector as direction

NOTE: to compute the previous location of a vertex just store the modelview matrix of the previous render pass
NOTE: for directional blurring you sample the motion vector and average the color values at those locations
NOTE: note we are doing object motionblur... motion blurring camera is different and quite easy (the blur kernel is constant)
Further reading (optional, more advanced that what is needed for HW): http://graphics.cs.williams.edu/papers/MotionBlurI3D12