Render to texture

You can render to a buffer that isn't shown immediately, but is stored in a texture for use in later rendering stages.

Application examples:

- "Real" environment mapping. Render the environent to a texture.
- Spatial filters
- Temporal filters
- Shadows

The heart of multi-pass rendering.

Render to texture

Can be done in different ways:

glReadPixels + glTexImage
Bad, goes over the CPU

glCopyTexImage
glCopyTexSubImage
Copying within VRAM

pBuffers (Pixel buffers)
Frame buffer objects (FBO) - preferred!
Write directly to another buffer.

Framebuffer objects, usage

Create a reference, just like for textures:

```
glGenFramebuffers(1, &fb);
```

Select a texture:

```
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex, 0);
```

Same for "renderbuffer":

```
glGenRenderbuffers(1, &rb);
glBindRenderbuffer(GL_RENDERBUFFER, rb);
glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24, width, height);
```

Framebuffer objects, usage

Select active FBO:

```
glBindFramebuffer(GL_FRAMEBUFFER, fb)
```

Turn off FBO, render to the usual frame buffer:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0)
```

Note! Old OpenGL code may add EXT-suffixes! Just remove them.

Debugging of FBO

While developing, you should catch FBO errors. This is made with glCheckFramebufferStatus.

Typical problems: Bad parameters to your output texture. Some implementations demand the output texture to be configured for closest neighbor interpolation.

It is harder to make an FBO work than you may think. Therefore, we have some packaged code in the lab material.

Example using glCopyTexSubImage

```
void display()
{
    glBindTexture(GL_TEXTURE_2D, minitexid);

    glViewport(0, 0, width, height);

// Draw what should go in the texture
    glClearColor(1,1,0.5,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawTextureScene();
    glFlush();

// Copy result to the texture
    glBindTexture(GL_TEXTURE_2D, tex);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, width, height);

    glViewport(0, 0, lastw, lasth);

// Render final image using the generated texture
    glClearColor(0.3, 0.3, 0.7,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawMainScene();
    glutSwapBuffers();
}
```

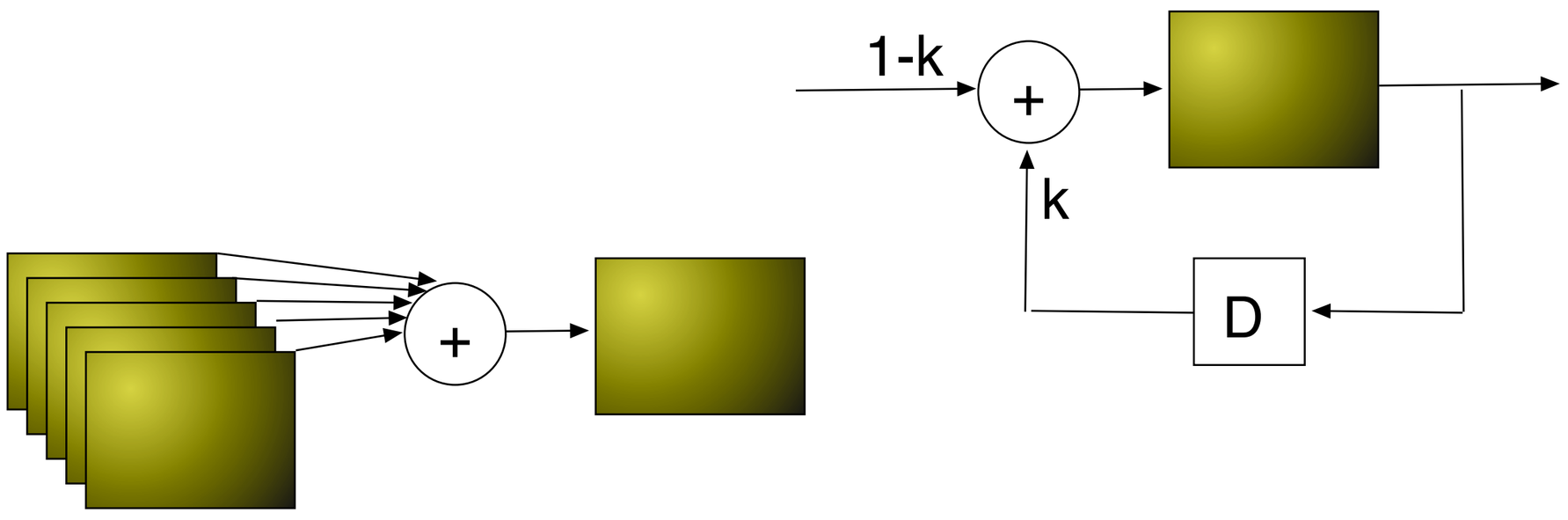
Example with FBO

```
void display()
{
// render to the FBO
    glBindFramebuffer(GL_FRAMEBUFFER, fb);
    glBindTexture(GL_TEXTURE_2D, minitexid);

    glViewport(0, 0, width, height);

// (draw something here, rendering to texture)
    glClearColor(1,1,0.5,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawTextureScene();

    glViewport(0, 0, lastw, lasth);

// render to the window, using the texture
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glBindTexture(GL_TEXTURE_2D, tex);

    glClearColor(0,0,0,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawMainScene();
    glutSwapBuffers();
}
```

Temporal filtering

Filtering in *time*!

Simple application of render-to-texture.

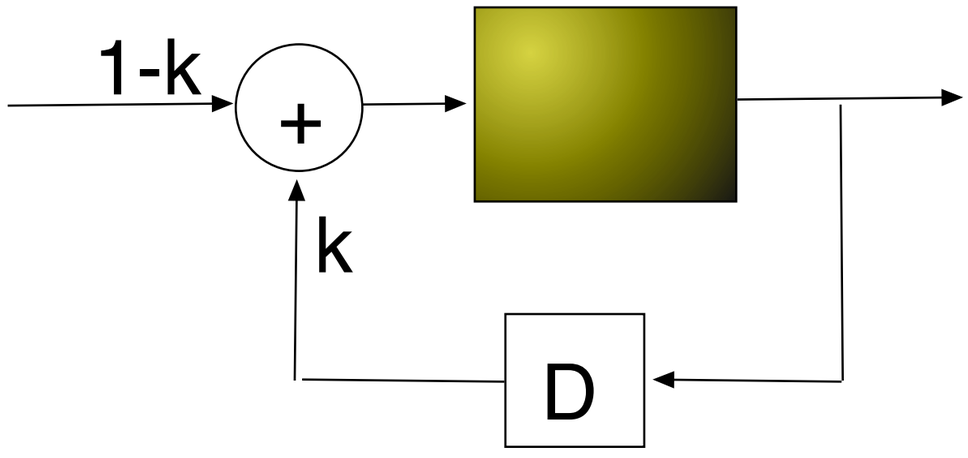


Recursive temporal filtering

Creates motion blur!

The picture is blended with an older one.

Only needs a single previous image.



Recursive temporal filtering on GPU

• Render the scene. Arbitrary contents plus an older copy in a texture buffer. Every pixel is weighted together with the older copy over the entire image.

If you use glCopyTexSubImage: Copy the resulting image to the older copy.

With FBOs: Use two FBOs, one as source, one as destination.

CPU code

Select the texture (RTF buffer)

```
glActiveTexture( GL_TEXTURE1 );
glBindTexture(GL_TEXTURE_2D, rtftex);
```

Render scene.

Copy texture:

```
glBindTexture(GL_TEXTURE_2D, rtftex);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0,
                    0, 512, 512);
```

GPU code

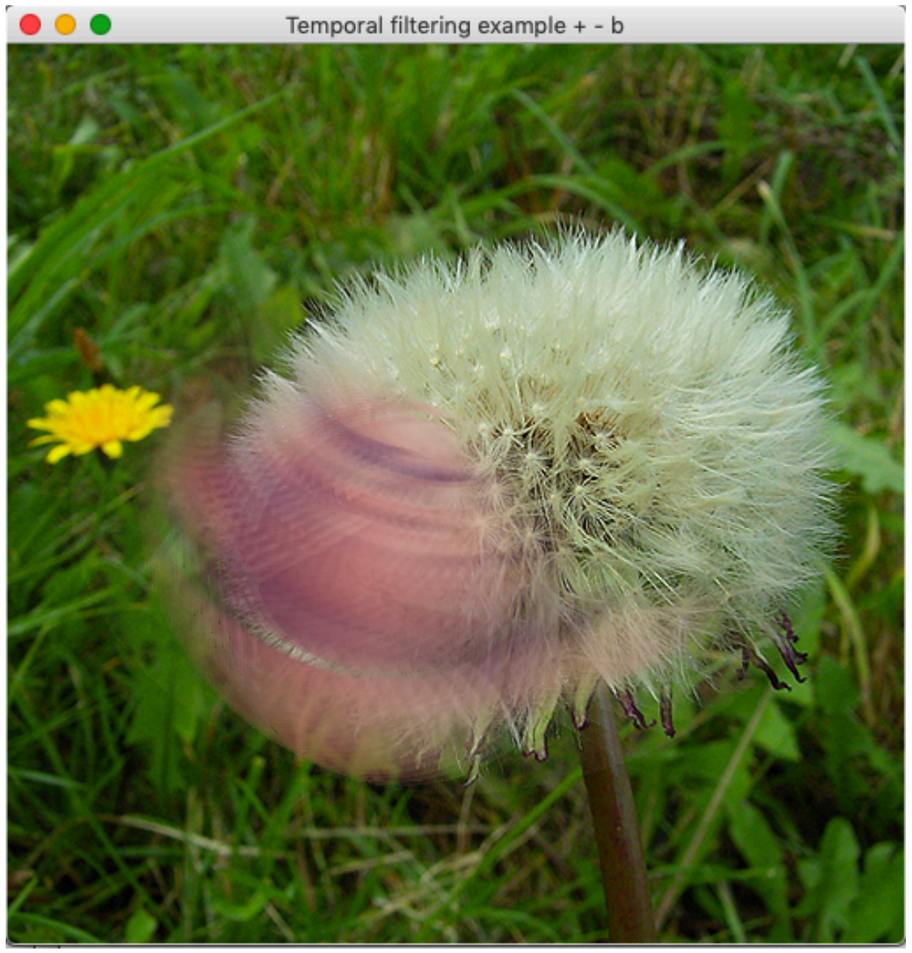
Vertex shader: Screen coordinated to texture coordinates!

```
out vec2 screenCoord;
screenCoord = vec2(gl_Position) / gl_Position.w / 2.0
    + vec2(0.5);
```

Fragment shader: perform blending between rendering result and texture:

```
outColor = texture(tex, texCoord) * (1.0-k) +
    texture(rtftex, screen- Coord) * k;
```

Movement blur



Processing particle systems with shaders

Large particle systems are best processed on the GPU. It can be done with shaders, or GPU Computing platforms like CUDA/OpenCL/Compute Shaders.

CPU: Only feasible for relatively small systems.

Particle system are suited for parallel processing.

Shaders are near the graphics, but less flexible than e.g. CUDA.

Minimal particle system

Position p
Velocity v

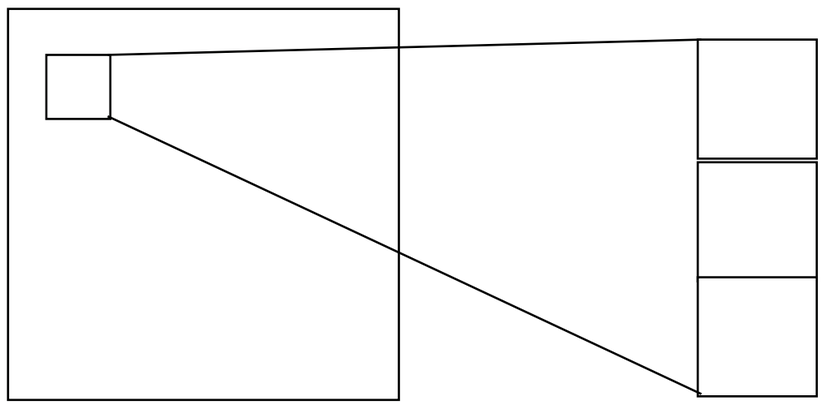
Update:

Position p = p + v * dt

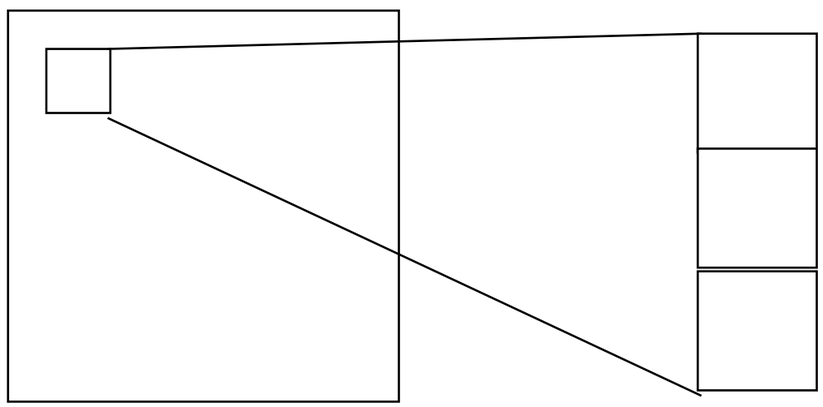
Velocity v = v + a * dt

Save data in textures!

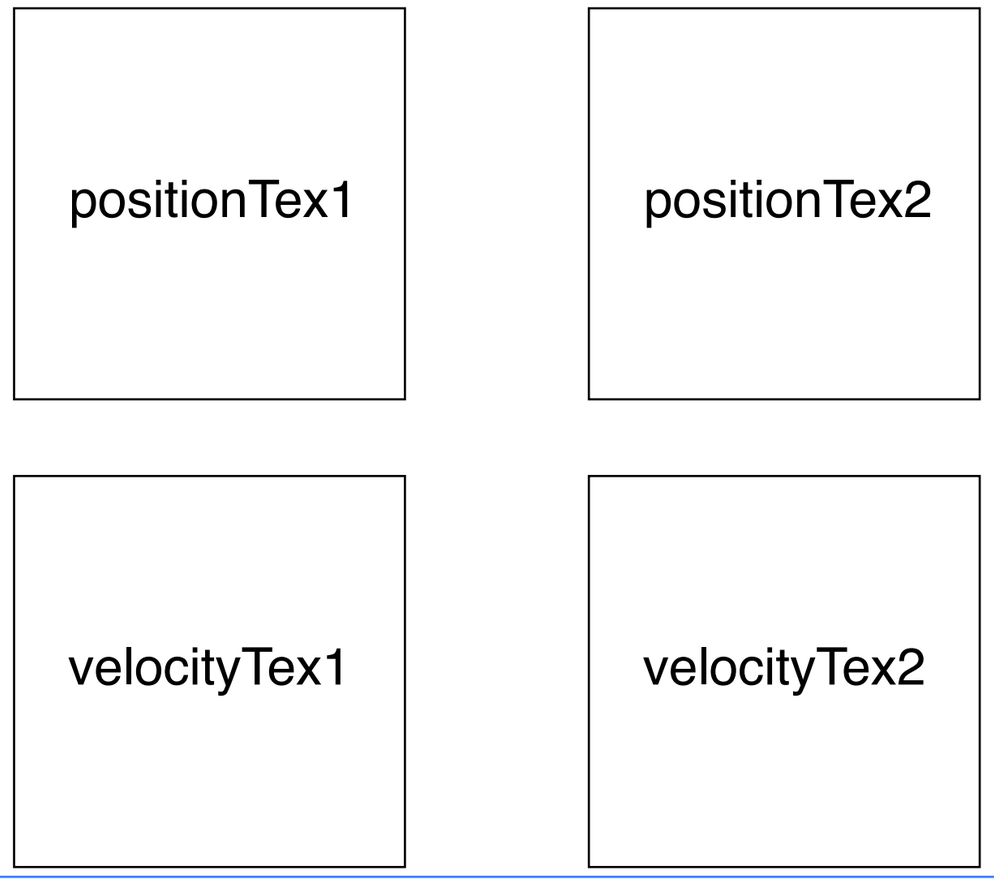
positionTex



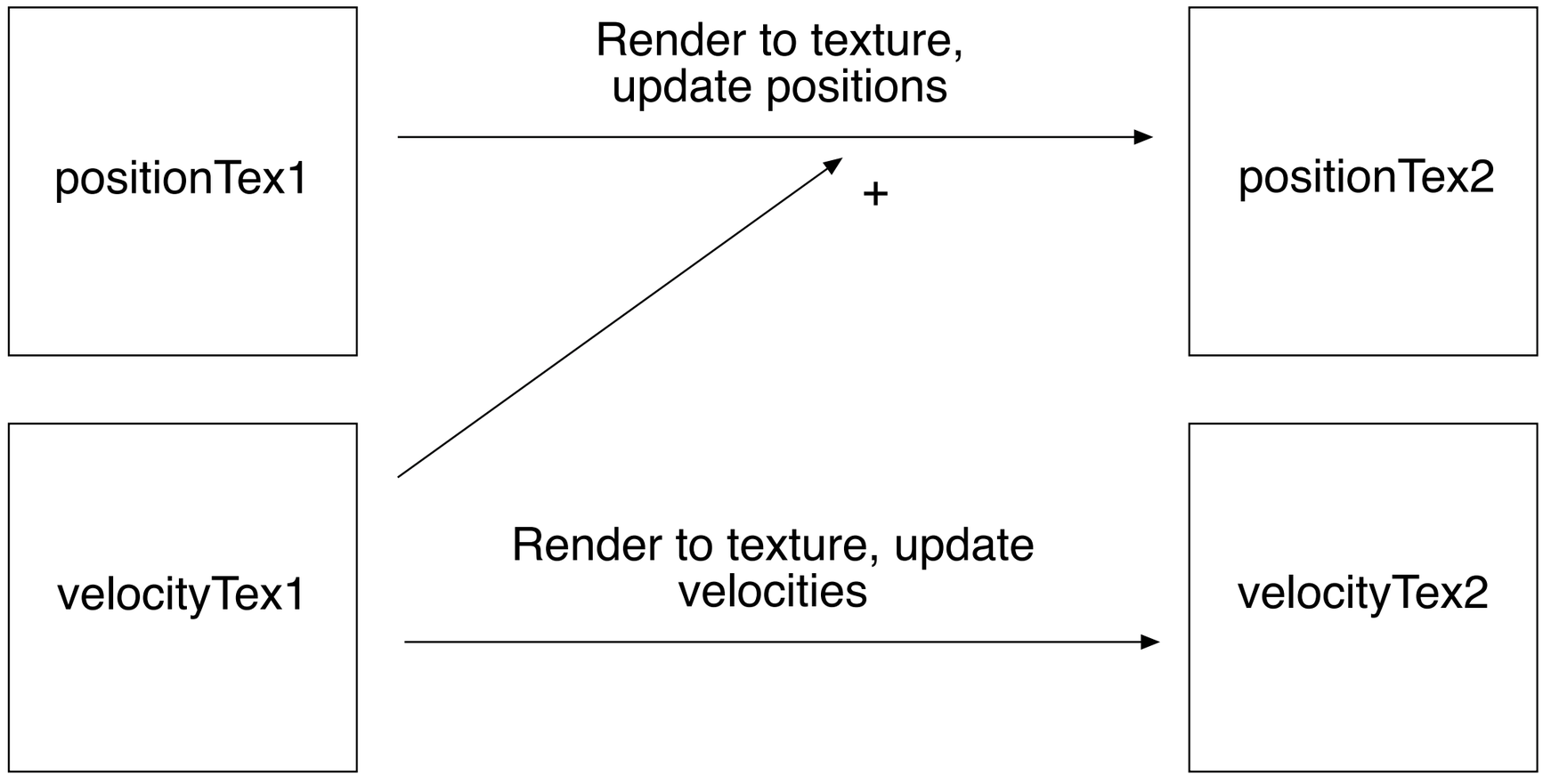
velocityTex



Double textures for ping-ponging



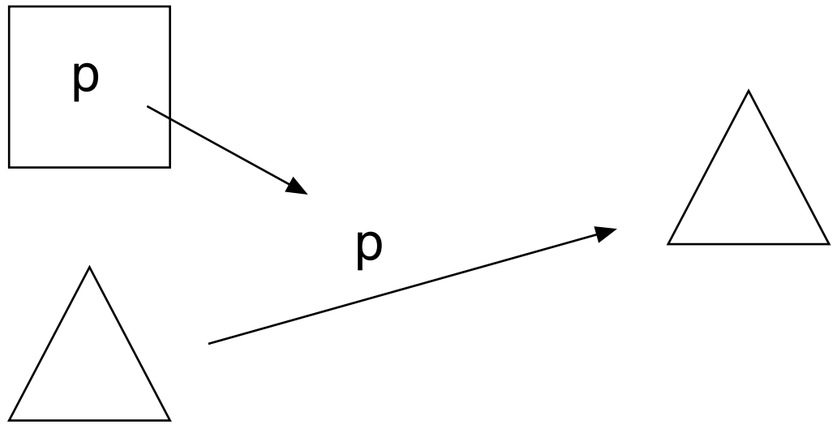
Render to FBO to update



Rendering by instancing:

Calculate the address (texture coordinate) in the texture from the instancing index

Set the position (vertex shader) for the billboard from the position read from the texture



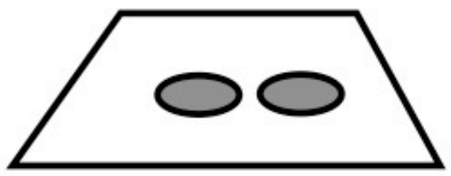
Collision detection for particle systems

- Within the system: Hard problem, many to many. Split to cells! Octrees is possible, but uniform subdivision is often more efficient.

- Test against the environment: Can be simplified, e.g with the Z buffer. (Somewhat beside the point for this course.)

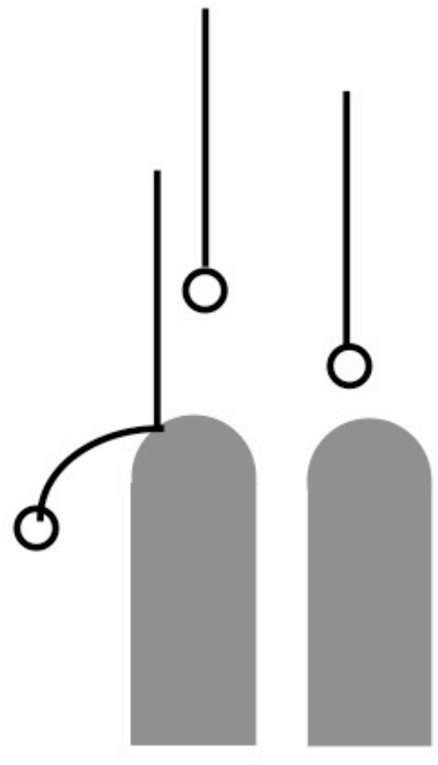
Example: Rain/snow etc

Camera for rendering Z-buffer from above



Z-buffer

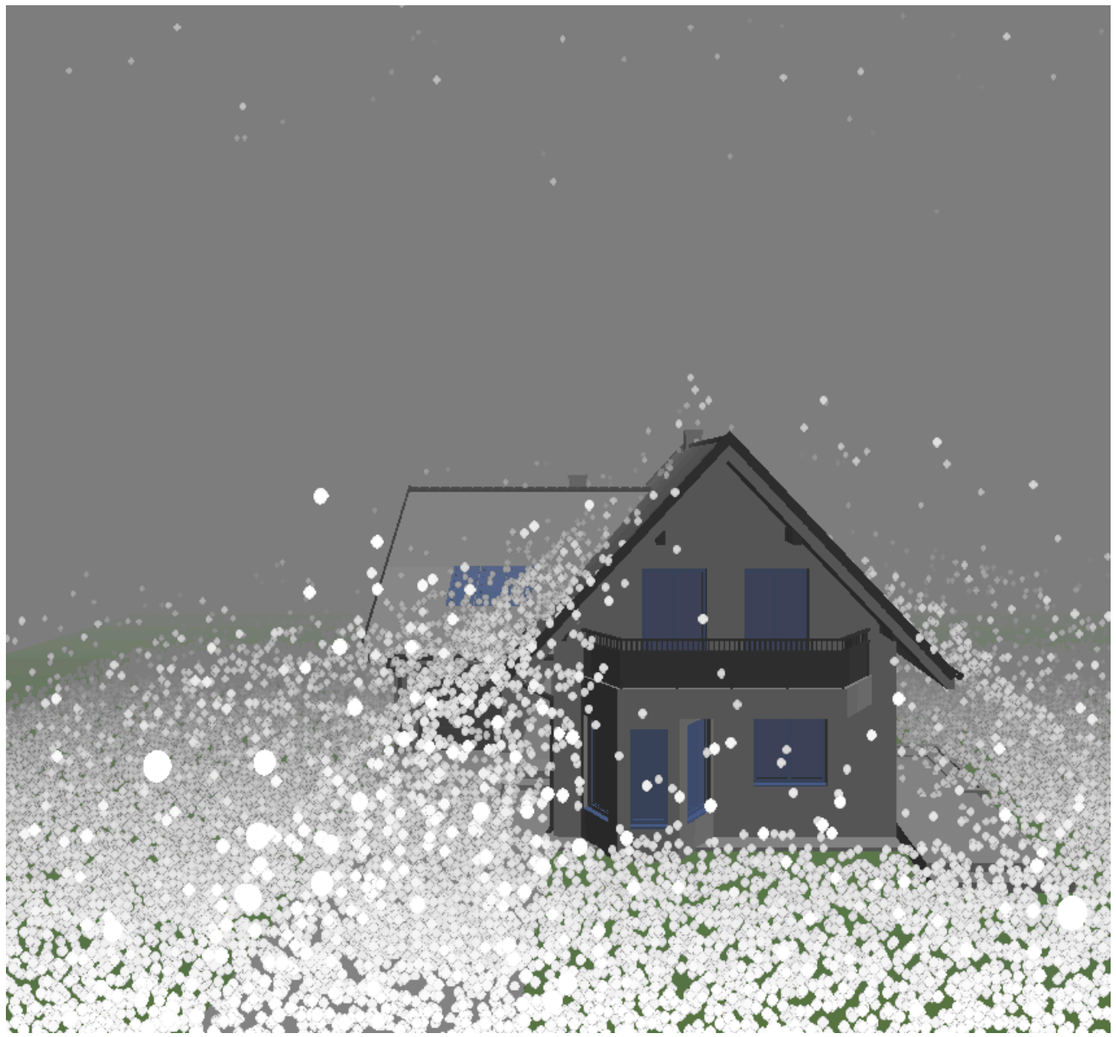
Scene



Particles can now bounce off objects only based on the Z buffer!

Project: Hailstorm

Particle system in shaders with instancing

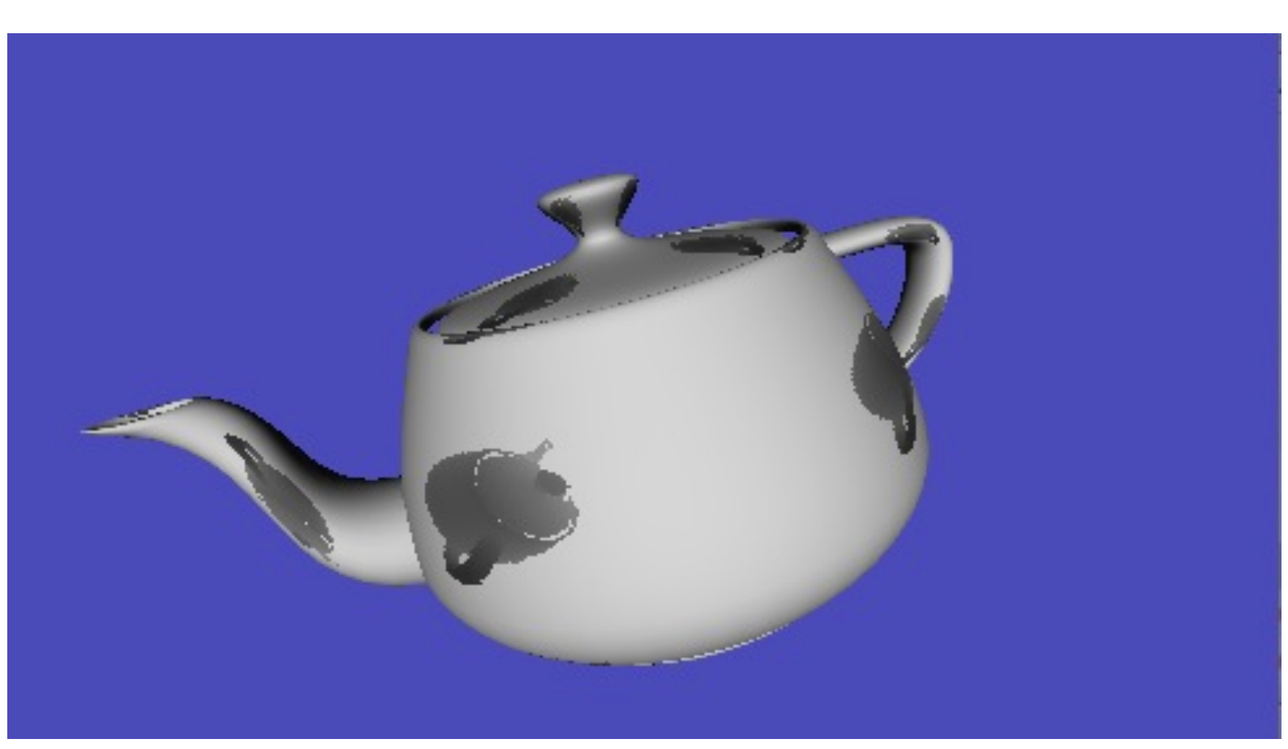


Collision with environment using the Z buffer

TSBK03
project by
Karl & Emma
2019

Z-buffer to texture

Can be done by copying between buffers or using FBOs.



Conclusions:

- Save particle data in textures, one per texel
- Bind textures to FBOs
- Update by rendering with ping-ponging
- Collision detektering
- Sorting (if needed)
- Render particles by instancing

Next time

Repetition (just in time before the dugga retake!)

Guest lecture by Stefan Gustavson!