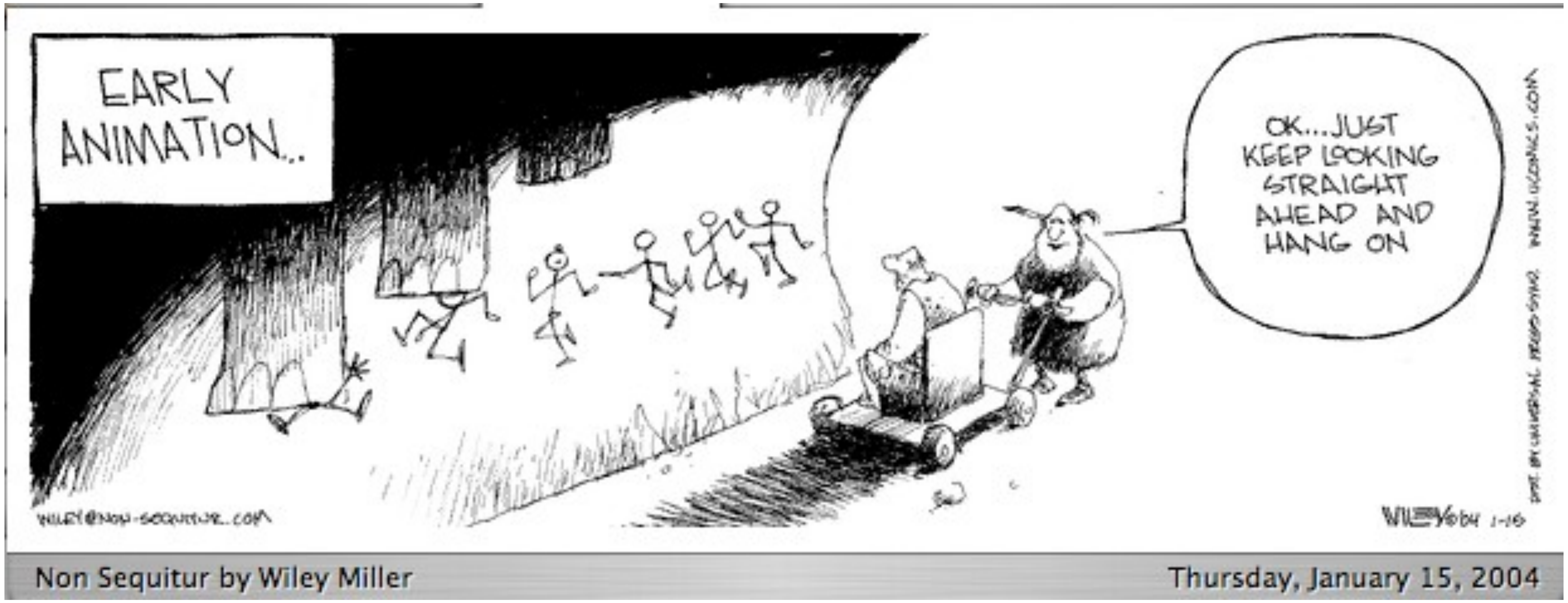
Animation

Essentially a question of flipping between many still images, fast enough



Animation as a topic

- Page flipping, double-buffering
- Sprite animation
- Movement and posing
- Collision detection and handling
- Deformations

Double buffering

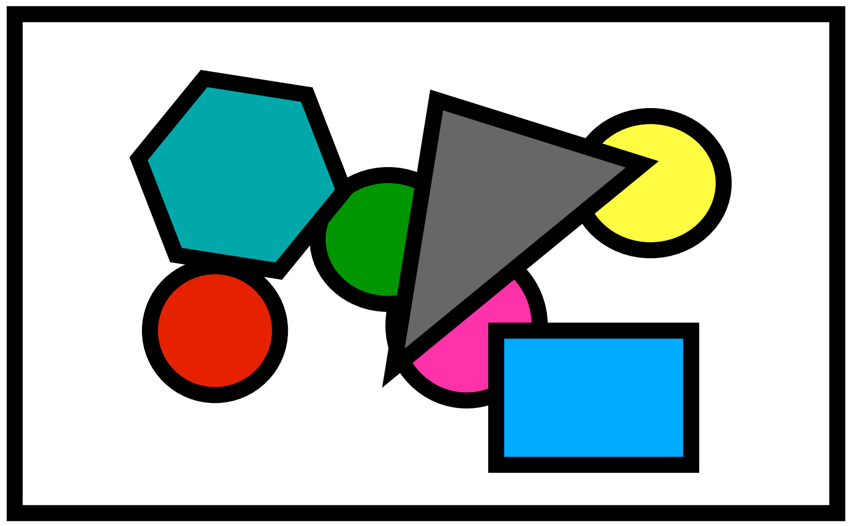
Flicker-free animation

The double buffering problem

When animating a scene with many objects in real time, it is not just a question of showing images:

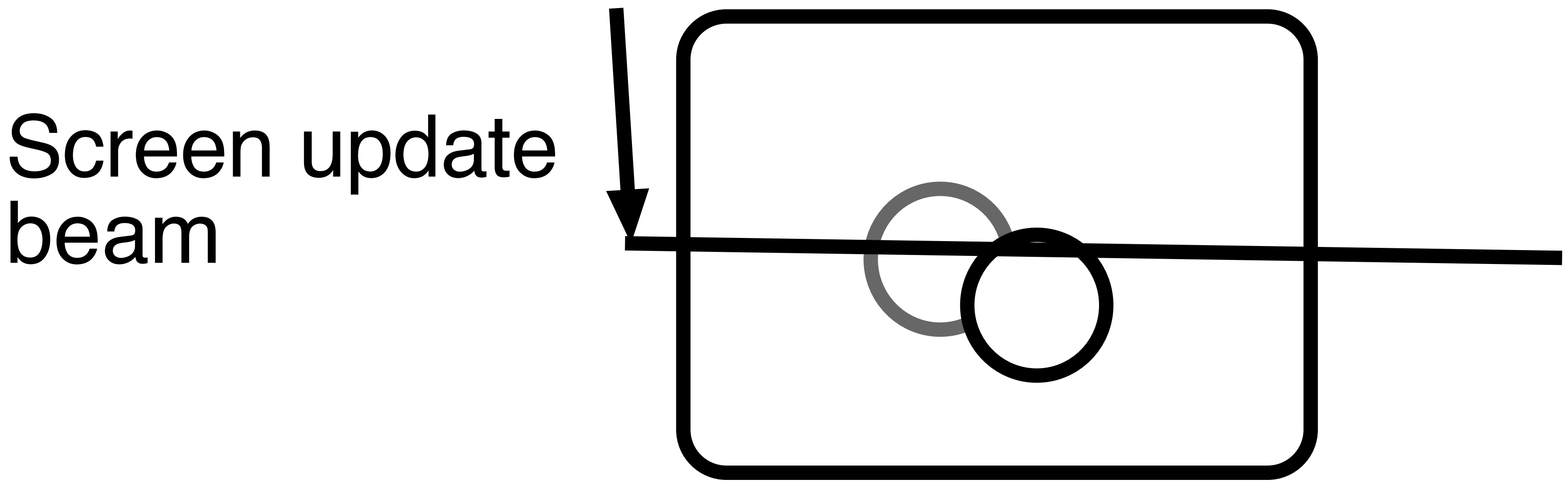
- **Erase the entire scene**
- **Draw each visible object in new positions**

This procedure may be visible if done on-screen!



Single buffered animation

Flicker



If the beam passes over an area while it is erased, *flicker* will occur.

Solutions

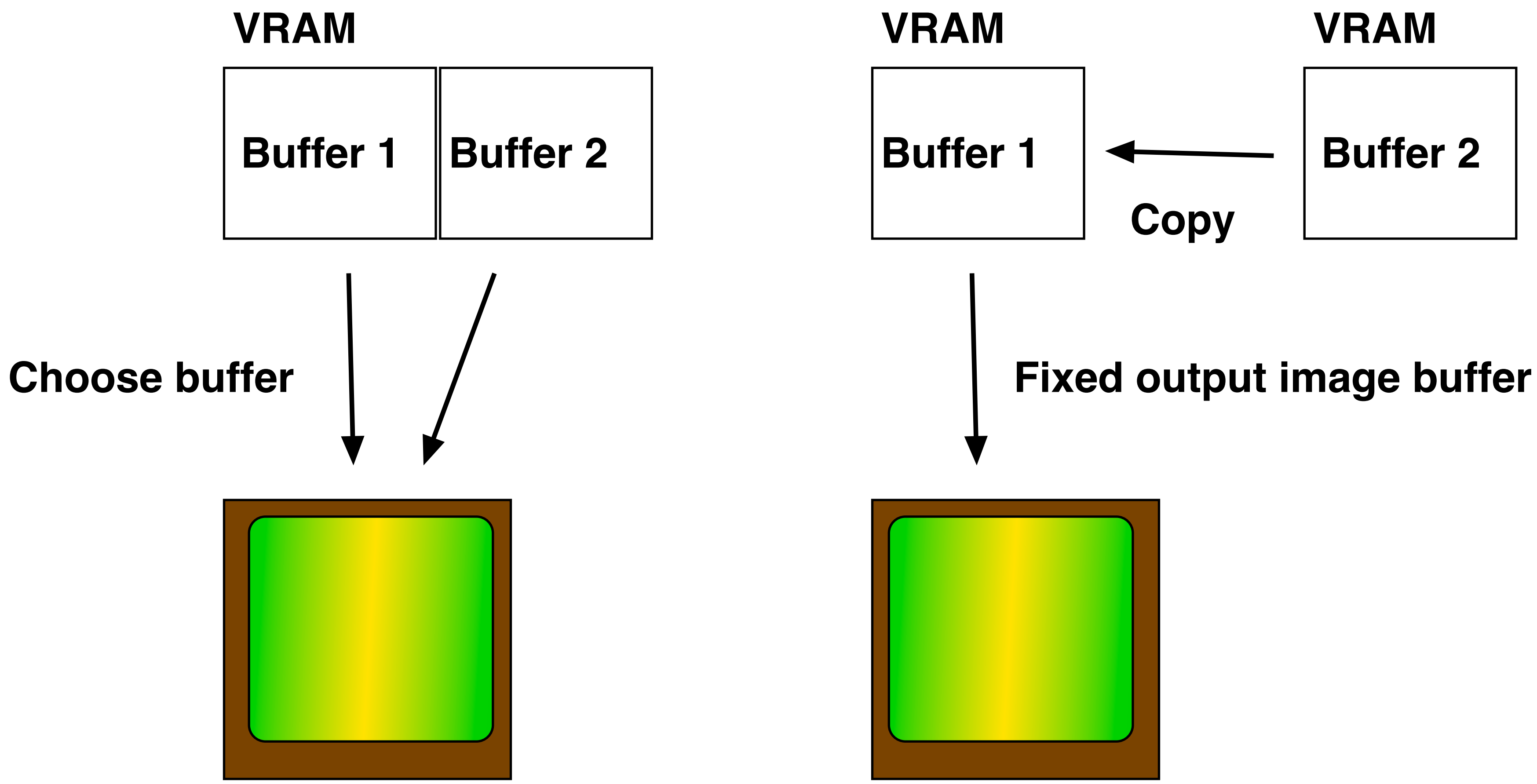
1) Don’t erase-and-redraw near the update beam

Unreliable.
Doesn’t work on all screens.

2) Double buffering.

Needs more memory. Otherwise easy to do and reliable.

Double buffering

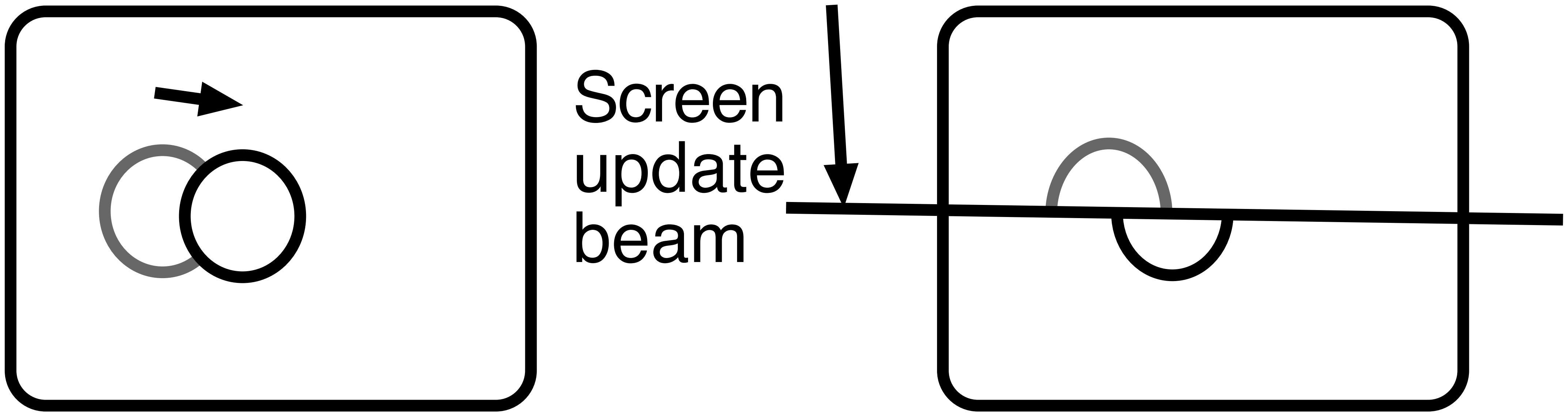


Animation with double output buffers

Animation with a single output buffer

Double buffered animation

Tearing



Occurs if buffers switch while the screen is being redrawn.

Synch with vsync to avoid.

Built-in VBL sync (vsync)

Modern systems have VBL sync built-in - even mandatory double buffering. You may need to turn ”vsync” off to test maximum frame rate.

Double buffering in OpenGL

Double buffer

- Pass GLUT_DOUBLE to glutInitDisplayMode
- glutSwapBuffers();

Repeated redraw

- glutRepeatingTimer() or timer callback with glutPostRedisplay()
- Update position variables

Sprite animation

2D animation based on 2D images.

Extremely common in games! (Often indie games and/or mobile games.)

Sprites in OpenGL

Use textured polygons with transparency! (Like billboards but without 3D.)

Special “blitter” calls existed in GL2, but they were not guaranteed to be fast!

Pseudo-3D effects

Scaled sprites on background with perspective:

Depth cue by **size**

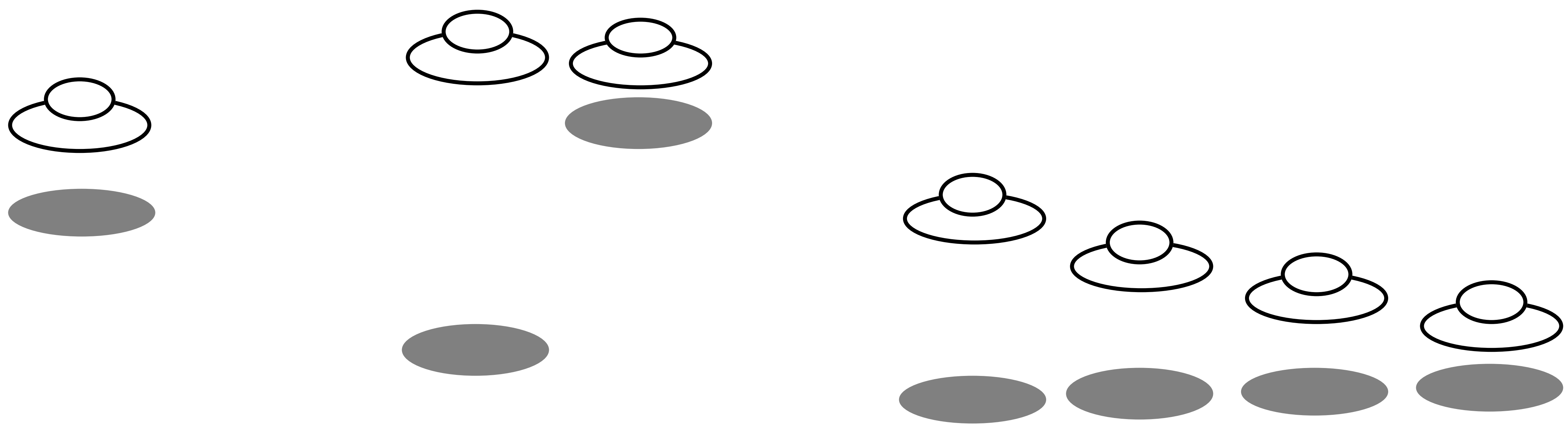
Side-scrolling with parallax scroll:

Depth cue by **movement**

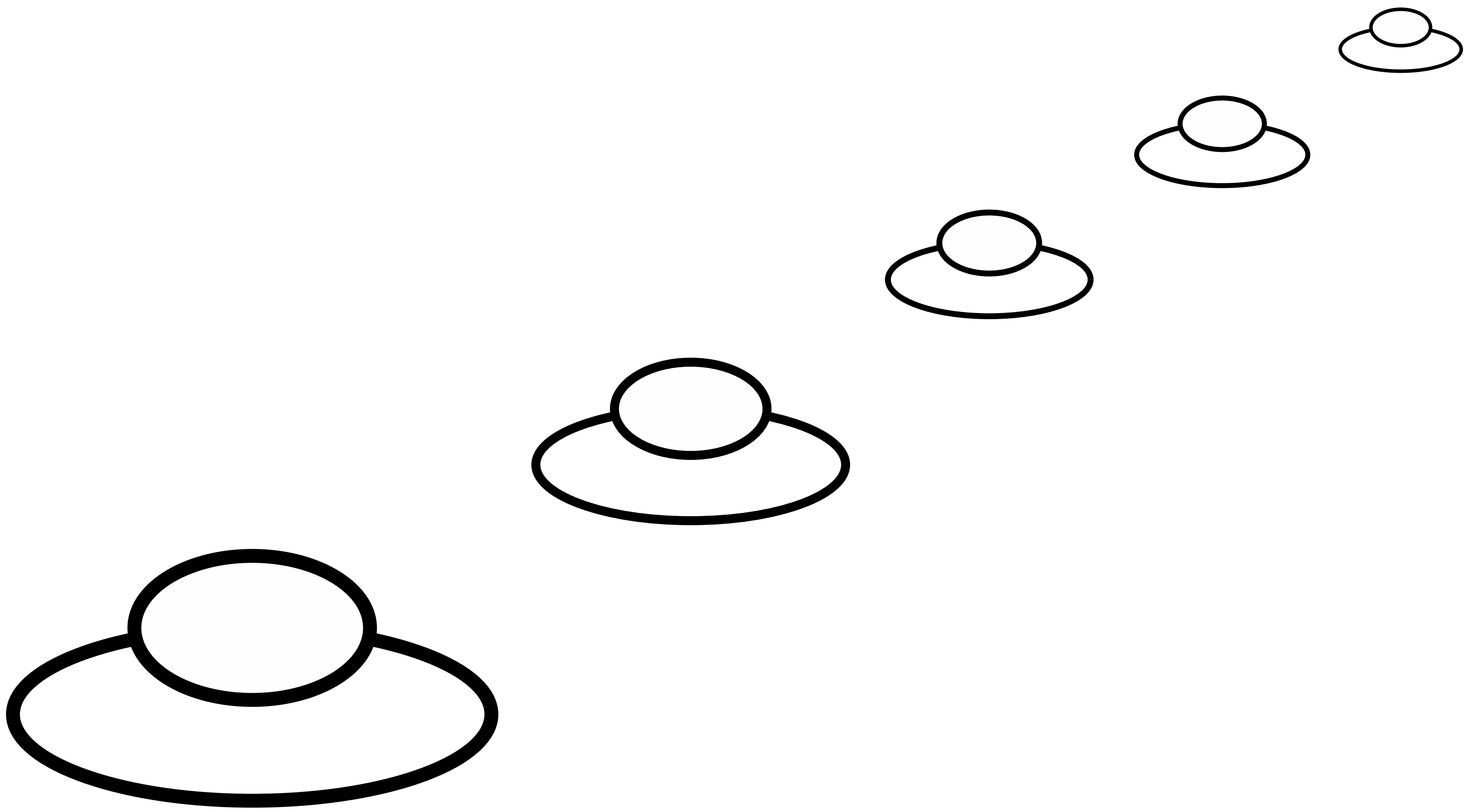
Depth due by **shadows**

Distance between object and shadow gives important information

Depth from shadows



Depth from size



Pseudo-3D effects vs 3D

- Depth from size = perspective projection

- Parallax scroll: Comes for free to some extent, but can be emphasized with cameras observing the viewer

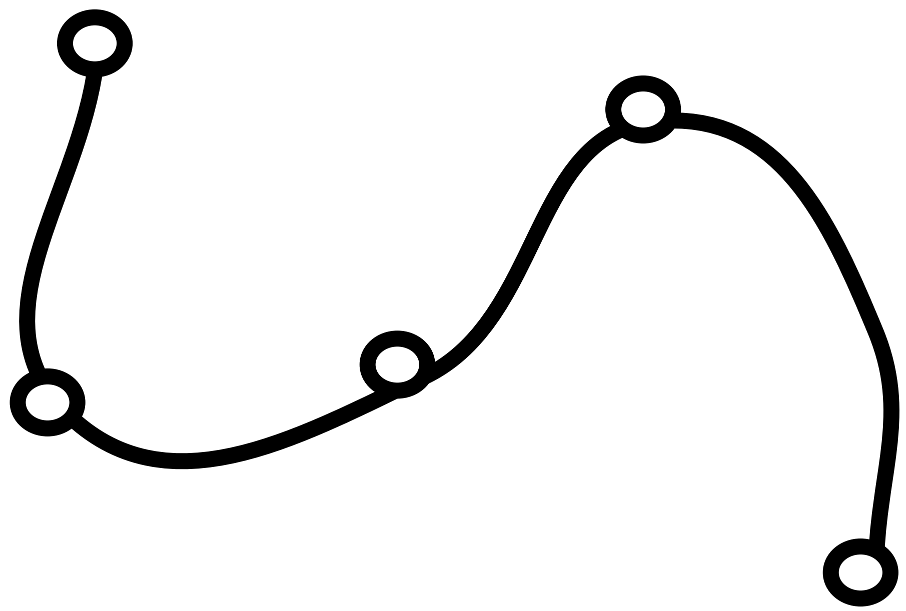
- Depth from shadows: That is why shadows are important in 3D! It is needed for ”full 3D” experience.

Animation techniques for moving objects

- Procedural animation
- Physics-based animation
- Pre-programmed animation paths

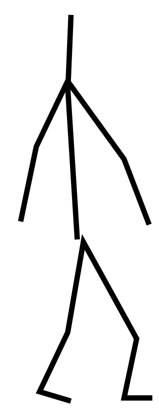
Animation paths

Use Catmull-Rom splines! Predictable, smooth, continuous!



Character animation

- Pre-defined poses
- Key-frame animation
- Forward kinematics
- Inverse kinematics
- Physics based animation
- Motion capture



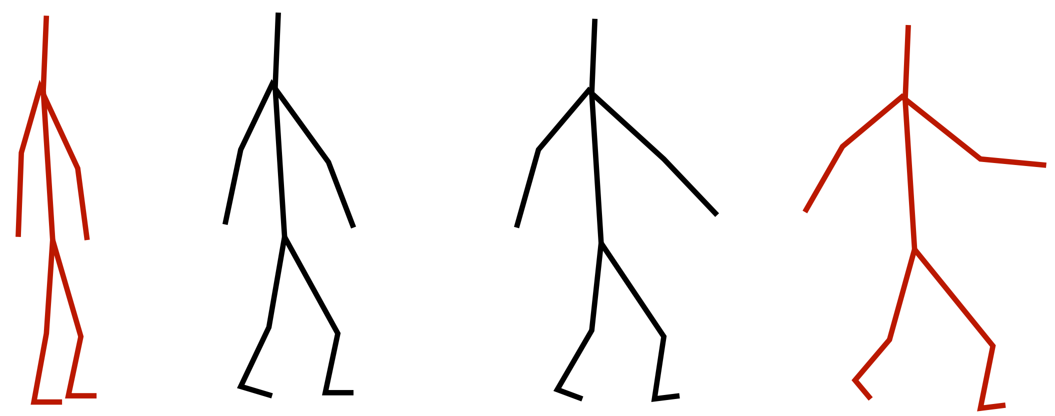
Key-frame animation

Pre-rendered animations

Key-frames are designed at suitable intervals

Frames between keyframes are interpolated (morphed)

Very common method for real-time animation



Kinematics

Kinematics = movement without forces

Forward kinematics:

Specify poses by specifying rotation of joints. Easy to implement, but specifying poses is much trial-and-error.

Inverse kinematics:

Goal-driven posing. Specify where some part should go (i.e. a hand) and calculate necessary rotations

Motion capture

Extremely common in movies!

- **Record by natural visuals only**
- **Tracking markers**
- **Active sensors on the body**

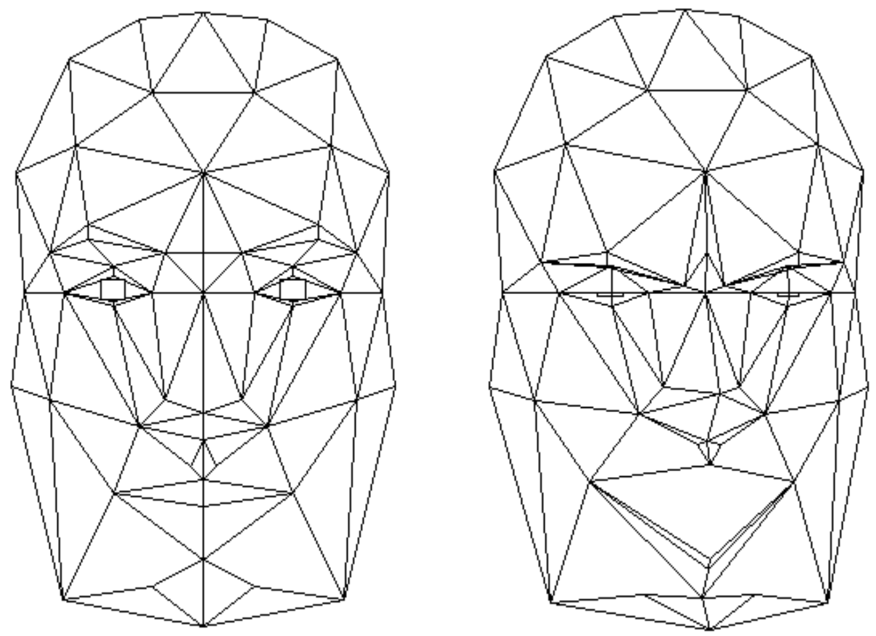
Perfect for pre-generated animations.

Face animation

Hard problem - we are very sensitive to errors!

Animate by action units (muscle based) or face animation parameters (extreme detail)

FAPs part of the MPEG-4 standard.



The Candide model

Some advanced animation topics

- Bones and skinning systems
- Deformations
- Physics-based animation
- Quaternions, SLERP

Mainly subjects for later courses

Particle systems

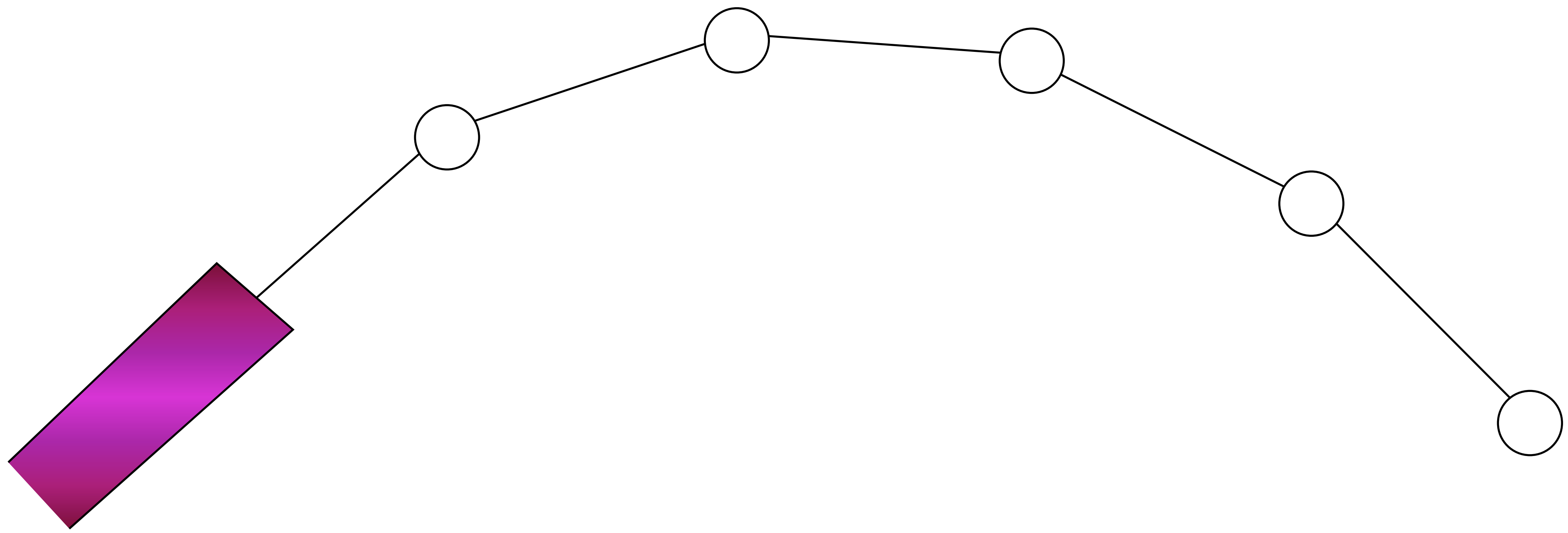
Spectacular effects with little effort!

Many small moving objects.

- **Explosions**
- **Water**
- **Fire**
- **Snow**
- **Rain**

Particle system

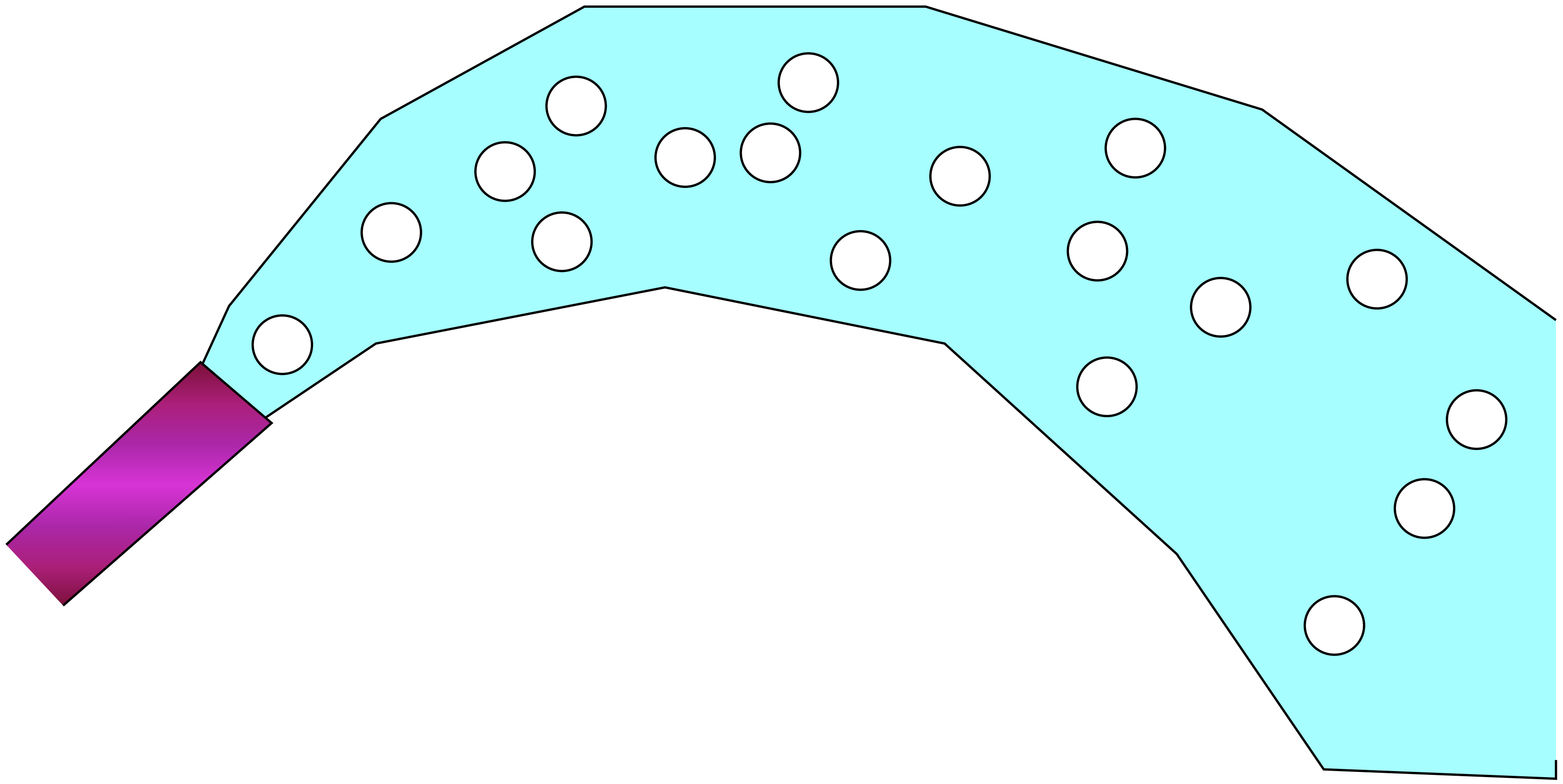
Example: Water



No randomness - bad

Particle system

Example: Water



Particle system

- Initial position
- Initial speed (usually with some randomness))

- Movement (usually independent, physically realistic)

- Termination rule (e.g. hits ground, fades away after some time...)

Particle system

Movement according to fundamental physics:

acceleration = gravity + forces/mass
speed = speed + acceleration
position = position + speed

“Euler integration”

Particle system on GPU

CPU-driven particle systems OK up to a certain size

Data transfer (new positions) of all particles can be a bottleneck

Can the whole particle system be computed on the GPU?

Texture based particle systems

Use textures to store x, y, z, dx, dy, dz

Store as color components (r, g, b)

Needs advanced texturing features (render to texture, floating-point buffers)

Particles as billboards. Each polygon must identify its particle data.

Separate compute kernels for particle systems

CUDA, OpenCL, Compute shaders

Free choice of data formats

Less integration with the OpenGL pipeline

Drawing particle systems

Large number of very simple models (billboards)!

Modest demands on GPU, but very large number of function calls!

Solution: Instancing

Instancing

Draw a large number of the same model!

Each instance has an index, the instance number.

glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 10);

draws a triangle 10 times!

gl_InstanceID tells the shader which instance we have. Use for affecting position.

Billboard instancing demo

One single call to glDrawArraysInstanced

Position trivially affected by gl_Instance ID

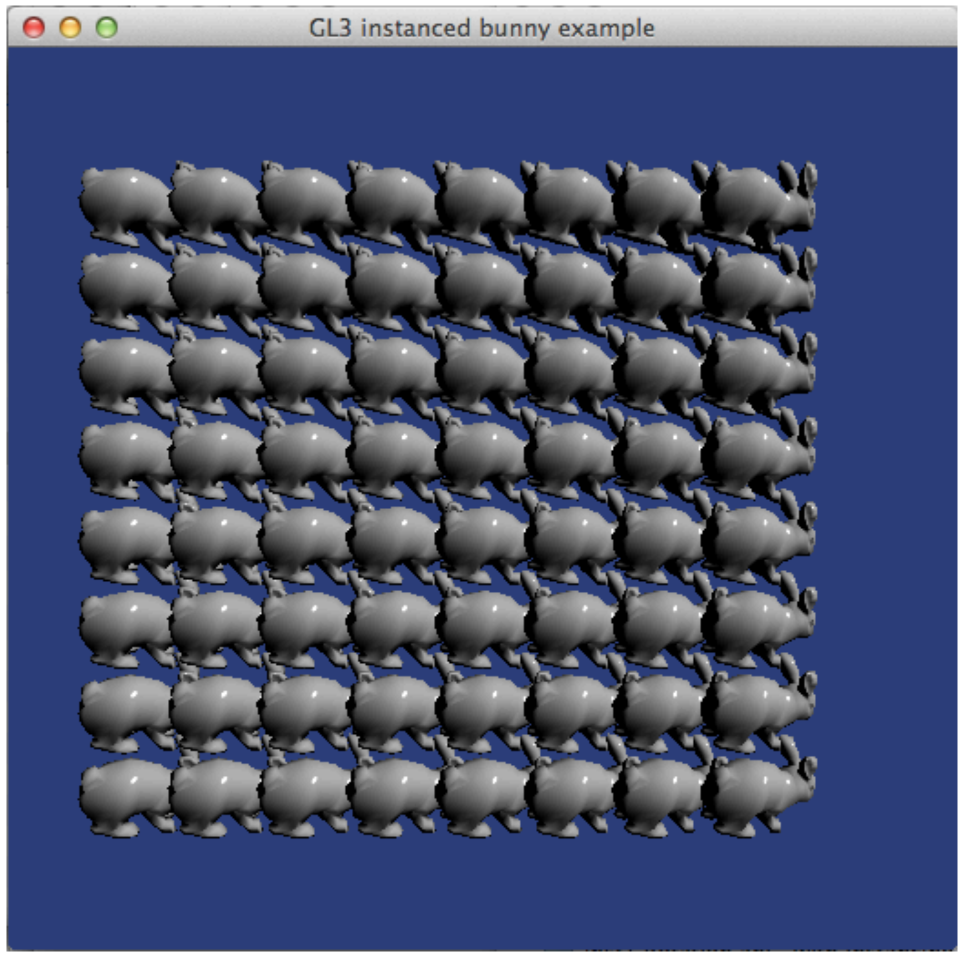
```
#version 150

in  vec3 in_Position;
uniform mat4 myMatrix;
uniform float angle;
uniform float slope;
out vec2 texCoord;

void main(void)
{
 mat4 r;
 float a = angle + gl_InstanceID * 0.5;
 float rr = 1.0 - slope * gl_InstanceID * 0.01;
 r[0] = rr*vec4(cos(a), -sin(a), 0, 0);
 r[1] = rr*vec4(sin(a), cos(a), 0, 0);
 r[2] = vec4(0, 0, 1, 0);
 r[3] = vec4(0, 0, 0, 1);
 texCoord.s = in_Position.x+0.5;
 texCoord.t = in_Position.y+0.5;
 gl_Position = r * myMatrix *  vec4(in_Position,
1.0);
}
```

Instancing complex models

Less significant; A more complex model puts enough load on the system to hide the impact of instancing.



Basic: Start on CPU
Advanced: Go for GPU acceleration

Performance is important, but GPU based particle systems are beyond basic course goals.