Game Graphics & Real-time Rendering

CMPM 163, S2019

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Manu Thomas (TA)
mthomas6@ucsc.edu

David Abramov (TA)
dabramov@ucsc.edu

Website: creativecoding.soe.ucsc.edu/courses/cmpm163_s19
Slack: https://ucsccmpm163.slack.com

Querying neighbors

- Conceptually, all fragments are processed simultaneously. Thus, when the fragment shader is in the midst of processing one pixel, it can not directly get information about any of the other fragments.

- However, in the fragment shader, if we are passing in a texture, then we can easily query the neighboring pixels in the texture (also called “texels”), as long as we know the width and height of the texture.

- This info is provided automatically by Unity as a special uniform float4 called **_MainTex_TexelSize**, where **.x** = 1.0/width, **.y** = 1.0/height, **.z** = width, and **.w** = height

Querying neighbors

- Or in general, as a special uniform float4 called _[NAME]_TexelSize, where [NAME] is the name of the property that defines any texture.

Querying neighbor

```
_MyTex("Texture", 2D) = "white" {}

uniform sampler2D _MyTex;
uniform float4 _MyTex_TexelSize;

//blur current pixel with pixel to the left
fixed4 frag(v2f i) : SV_Target {
    float2 t = float2(MyTex_TexelSize.x, _MyTex_TexelSize.y);
    float4 currentVal =  tex2D( _MyTex, i.uv );
    float4 leftVal = tex2D(  _MyTex, float2(i.uv.x - t.x, i.uv.y ) );
    return lerp(currentVal, leftVal, 0.5);
}
```

Render to texture

A very powerful technique used to create a wide range of visual effects is to render the output of a shader program to an **off-screen buffer** (also called a “frame buffer object” or “FBO” or “RenderTexture”) rather than to the display. You can use this buffer to apply multiple rendering passes to your scene (or parts of your scene).

Render to texture

```
Renderer r = GetComponent<Renderer>();
//existing texture
Texture2D inputTex = Resources.Load<Texture2D>("Textures/t01");
//define off screen buffer to hold color data
RenderTexture rt = new RenderTexture(w, h, depthBits, RenderTextureFormat.RGBA32);
//set active shader
r.material.shader = myShader;
//execute myShader with access to inputTex, store output in a RenderTexture named rt
Graphics.Blit(inputTex, rt, r.material);
//create new empty texture to hold color data
Texture2D newTex= new Texture2D(w, h, TextureFormat.ARGB32, false);
//copy offscreen buffer into this texture, which can now be used for another shader pass
Graphics.CopyTexture(rt, newTex);
```

Examples

- An in-game “Security Camera”

  Output of camera is written to a texture, which is then used to decal a GameObject (lots of tutorials online, eg:

  https://www.youtube.com/watch?v=EdFg1NSVuQY

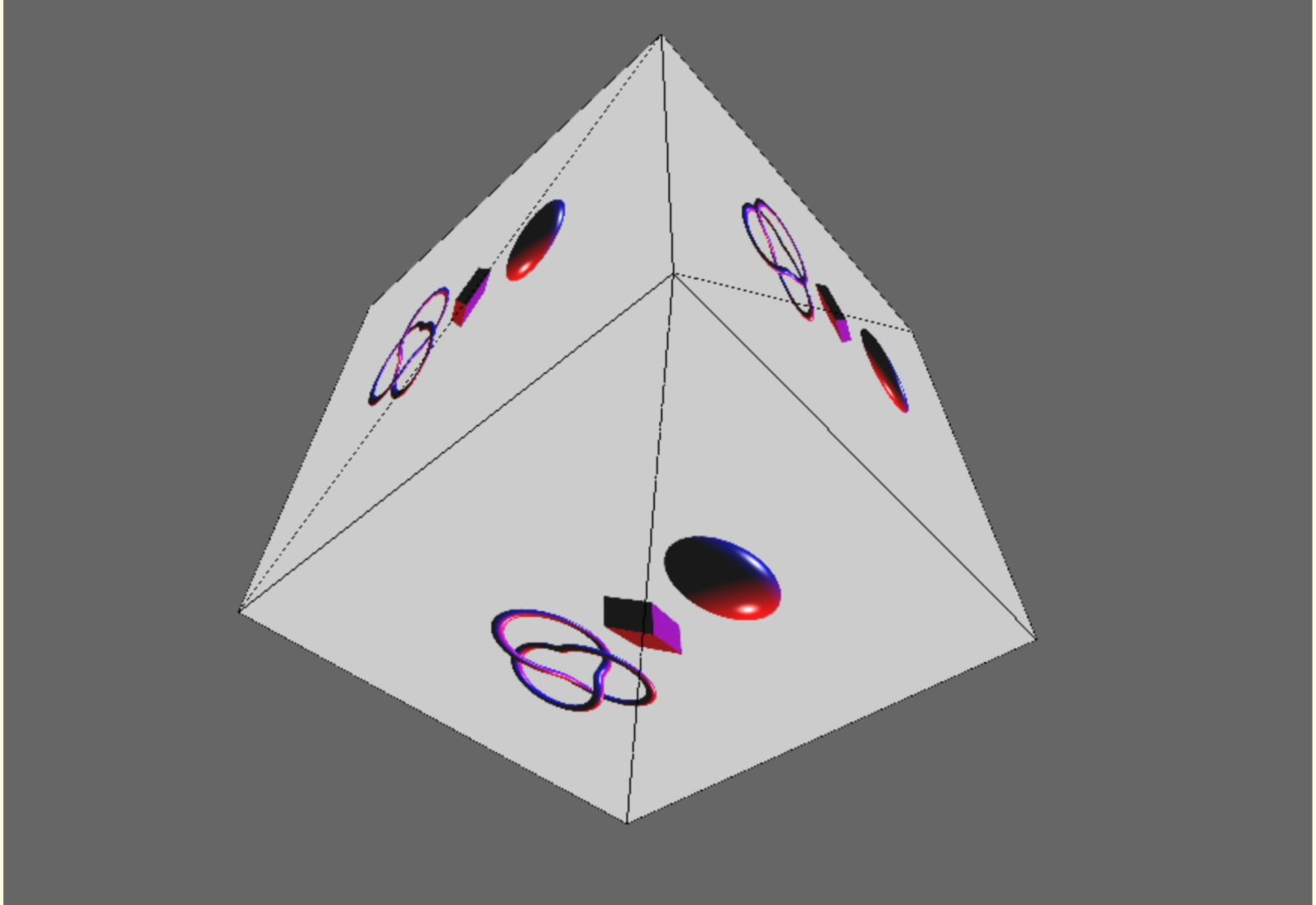
  https://www.youtube.com/watch?v=I3TE0hCw4Vg

- Cellular Automata or Conway’s Game of Life

  “Ping-ponging” of texture data is used to compute an emergent system by querying neighbor texels

- Reaction-Diffusion system

  Uses floating point textures and ping-ponging technique to create a simulation how a chemical substance spreads throughout a medium and reacts to other substances



“Ping pong” textures

- Building off of the render to texture example, we can use our shaders to perform general computation using our texture data.
- Within HLSL, textures are read-only, and we can’t write directly to our input textures, we can only “sample” them.
- Using the render to texture target, however, we can use the output of the shader itself to write to a texture.
- Then, if we can chain together multiple rendering passes, the **output** of one rendering pass becomes the **input** for the next pass.
- In the ping-pong technique, we create a kind of simple ring buffer, where after every frame we swap the off-screen buffer that is being read from with the off-screen buffer that is being written to.
- We can then use the off-screen buffer that was just written to, and display that to the screen.

“Ping pong” textures

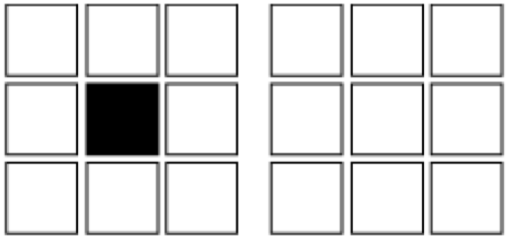
- We can then use the off-screen buffer that was just written to, and display that to the screen.
- This provides with a way to perform iterative computation on the GPU, which is useful for simulating a range of effects, such as smoke, water, clouds, fire, etc. (which we’ll explore in the coming weeks).
- In the Cellular Automata example, I show how we can use this technique to make a GPU version of Conway’s Game of Life: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- The Game of Life is a simple simulation that shows how even very simple rules can give rise to interesting, complex, emergent behaviors.

Game of Life

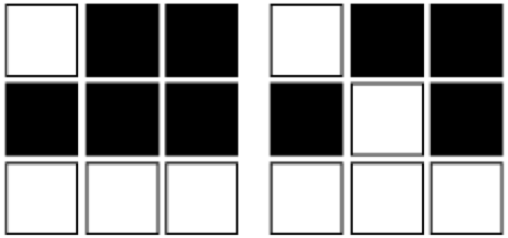
The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*, or "populated" or "unpopulated". Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

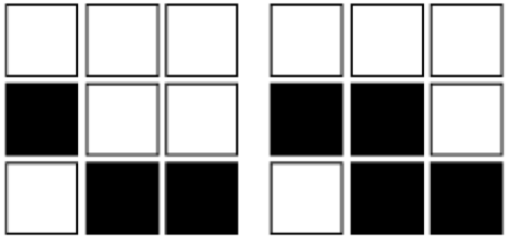
Game of Life



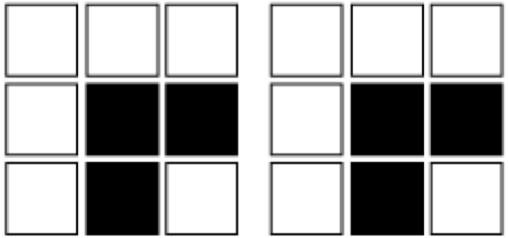
Loneliness
A cell with less than 2 adjoining cells dies.



Overcrowding
A cell with more than 3 adjoining cells dies.



Reproduction
An empty cell with more than 3 adjoining cells comes alive.



Stasis
A cell with exactly 2 adjoining cells remains the same.

Unity Texture manipulation

SetPixels, GetPixels (CPU, slow)

Puts data into or retrieves data from the Texture. The texture data must be available on the CPU in order to be retrieves. The texture data needs to be uploaded to the GPU in order to be rendered & used by shaders, etc.

Apply (CPU --> GPU, slow)

Copy data from the CPU to the GPU

CopyTexture (GPU, fast)

Moves Texture data from one Texture into another

ReadPixels (GPU --> CPU, slow)

Copies texture data from the GPU to the CPU

Blit (GPU, fast)

Triggers rendering into an off-screen buffer (a RenderTexture)

Texture formats

There are LOTS of different formats for texture data. (See https://docs.unity3d.com/ScriptReference/TextureFormat.html)

In this class we will mainly be using RGBA32, which is used for color. Each channel (red, green, blue, alpha) stores a 8-bit integer (0 – 255). For some of the simulations we will use RGBAFloat, which gives us full precision floating point decimals for each of the four channels. On the GPU itself, data is stored as a big 1D array, and the TextureFormat tells our shader how to access that array.

Other formats are used for depth information (which will discuss next class), compression, for HDR colors, and to support a range of video formats, etc.

Ultimately, we need to display to the screen, which expects 8-bit RGBA values.

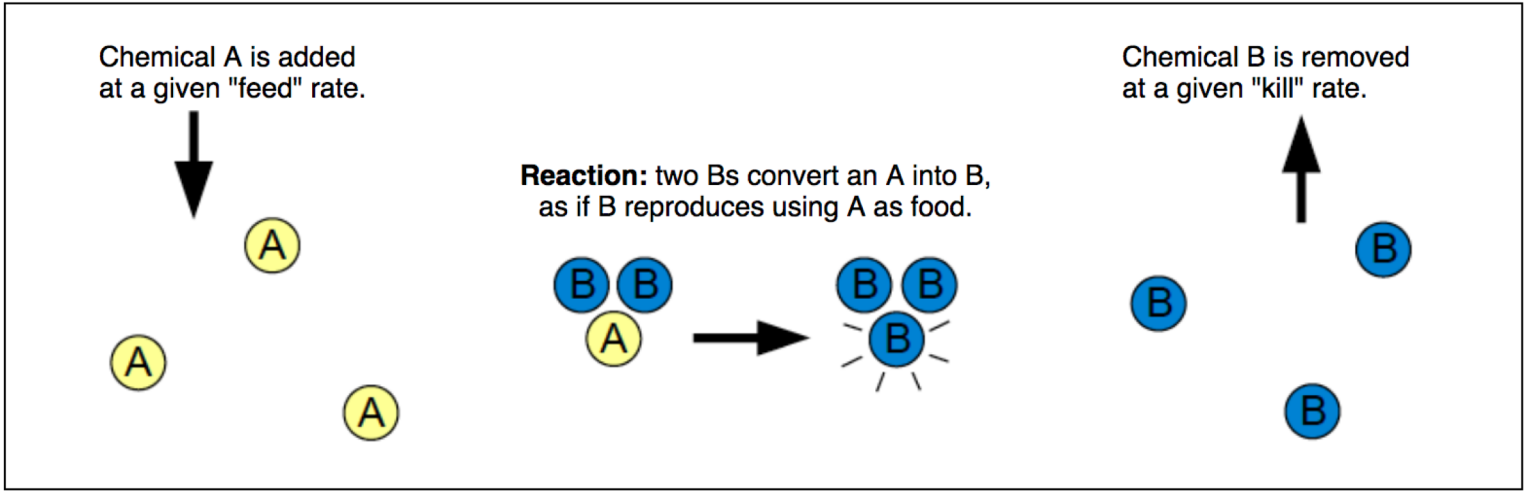
Reaction-Diffusion

"Reaction–diffusion systems are mathematical models which correspond to several physical phenomena: the most common is the change in space and time of the concentration of one or more chemical substances:
local chemical reactions in which the substances are transformed into each other, and diffusion which causes the substances to spread out over a surface in space.

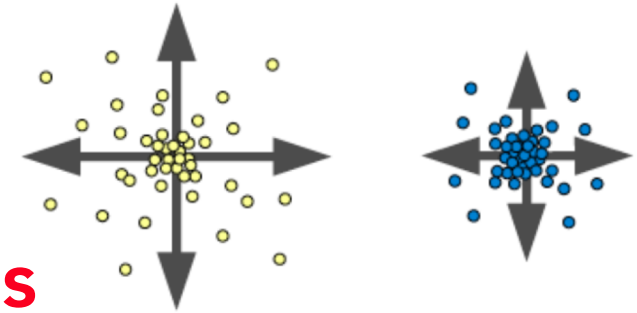
Reaction–diffusion systems are naturally applied in chemistry. However, the system can also describe dynamical processes of non-chemical nature. Examples are found in biology, geology and physics (neutron diffusion theory) and ecology."

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u),$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v.$$

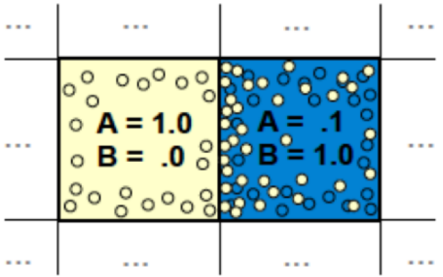
A simulation of two virtual chemicals reacting and diffusing on a 2D grid using the Gray-Scott model



Diffusion: both chemicals diffuse so uneven concentrations spread out across the grid, but A diffuses faster than B.

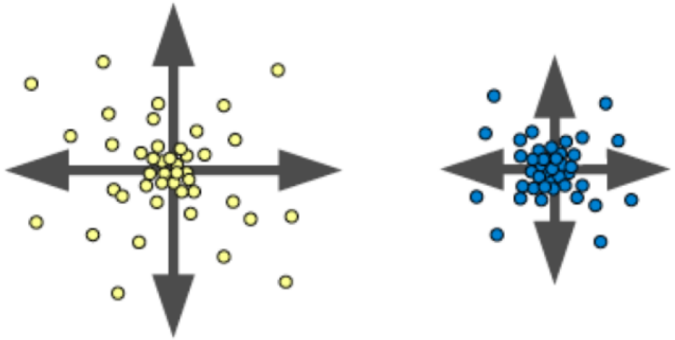


The system is approximated by using two numbers at each grid cell for the local concentrations of A and B. (The particles are not individually simulated.)

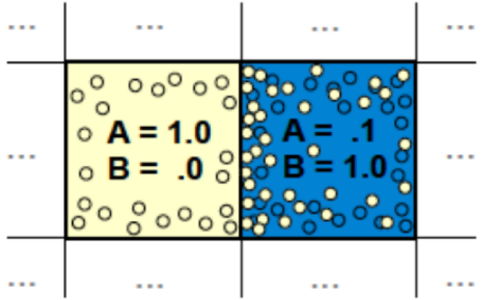


Karl Sims

Diffusion: both chemicals diffuse so uneven concentrations spread out across the grid, but A diffuses faster than B.



The system is approximated by using two numbers at each grid cell for the local concentrations of A and B. (The particles are not individually simulated.)



When a grid of thousands of cells is simulated, larger scale patterns can emerge.

The grid is repeatedly updated using the following equations to update the concentrations of A and B in each cell, and model the behaviors described above.