WebGPU

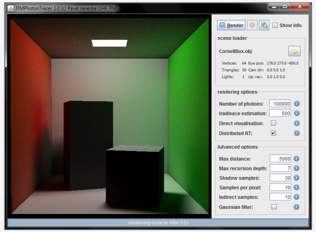
Compute shaders on the web

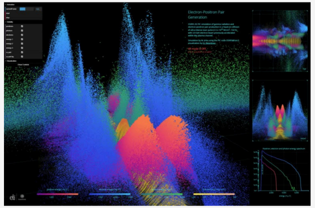
Pavel Janečka ~ PGRF 2024

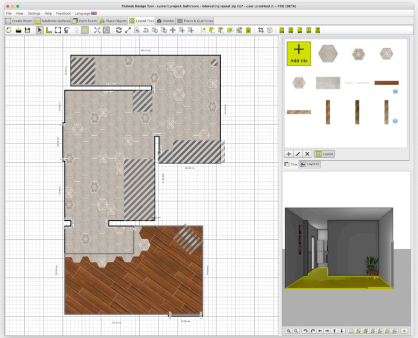
About me

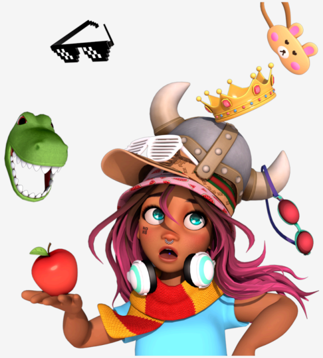
- UHK FIM
  - Bc. & Ing. (2009)
  - PhD (unfinished)

- Projects and teams of all sizes and shapes
- 2D, 3D, VR
- Web ❤️









History

- Native plugins (late 90’s - early 00’s)
  - VRML, X3D
- Adobe Flash (2005 - 2020)
  - Away3D, Stage3D
- WebGL 1 (2011)
  - Based on OpenGL ES 2.0 (2007)
- WebGL 2 (2017)
  - Based on OpenGL ES 3.0 (2012)
- WebVR (2016) / WebXR (2018)
- WebGPU (2023)

WebGPU

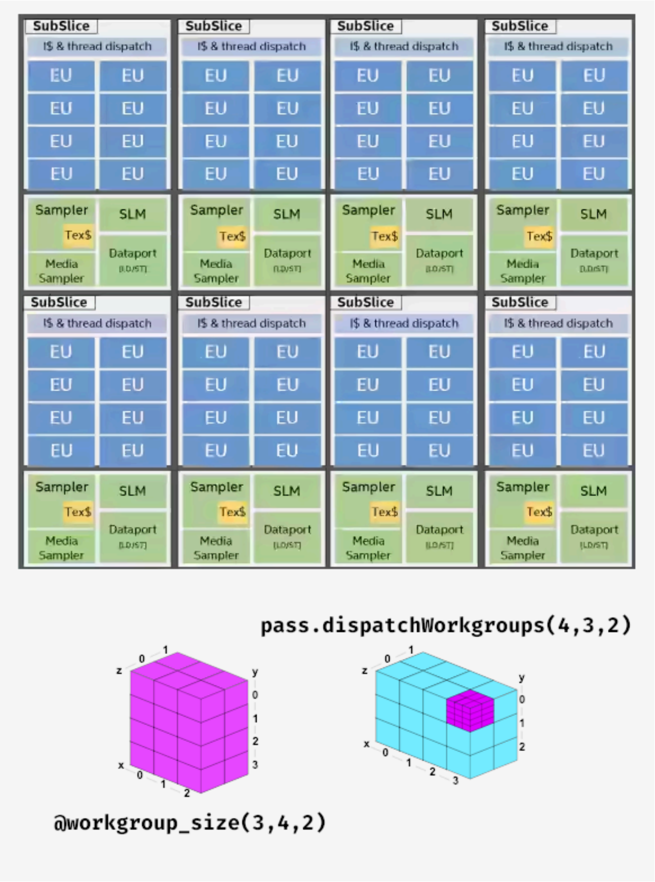
- Brand new low level API
  - Inspired mainly by Metal & Vulkan
  - Lean into asynchronicity of the web
  - Supported by all main browsers
- New WGSL ("wig-sil" / "wig-sal") shader language
  - Similar to Rust
  - SPIR-V compilers already exists
- Realtime rendering & GPGPU
  - Vertex / fragment / compute shaders
- Dawn vs. wgpu vs. WebGPU headers

WebGL2 vs. WebGPU

- No more global state 🙏
- Stateless / immutable API
- Completely asynchronous (even error model)
- Native support for video frame processing / sources
- Application portability by default (reasonable base limits)
- No more canvas management hand holding ⇒ unlimited canvas count
- Helpful error messages
- Index is the kind, beware of data type paddings
- Immutable textures and buffers
- Z Clip space -1, 1 ⇒ 0, 1

Compute shader

- Generic function with inputs and outputs
- Extreme parallelism (hundreds of GPU cores)
  - Slices ⇒ subslices ⇒ execution units ⇒ SIMT cores
  - Memory access is very costly
  - EUs are oversubscribed with work



WGSL compute shader

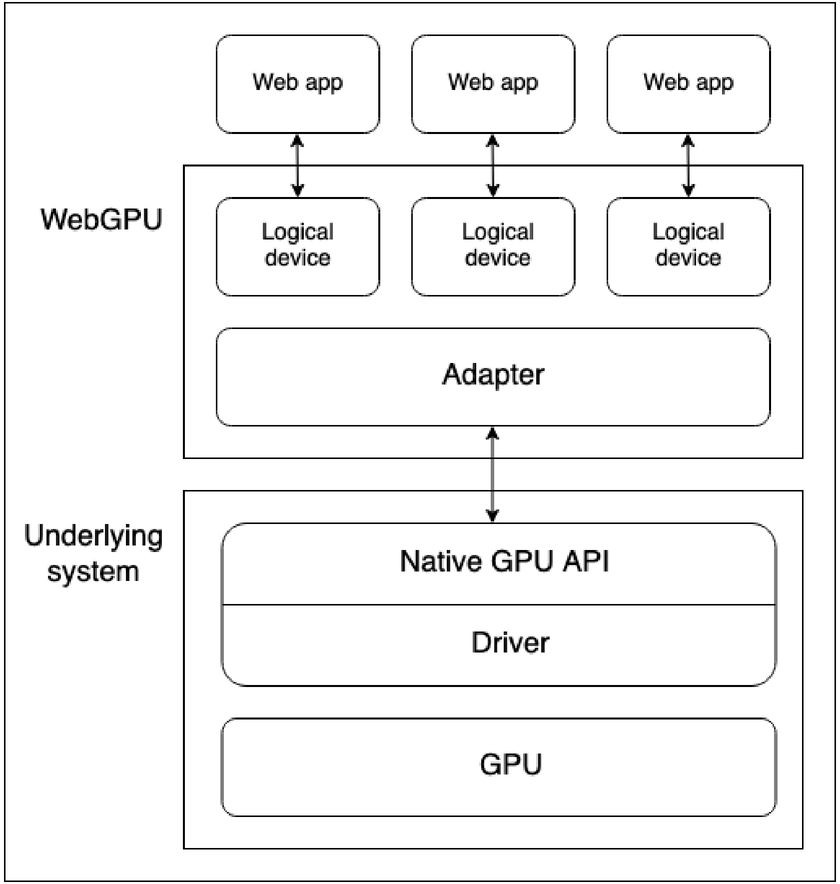
```
@workgroup_size(x, y, z)
```

Compute pass descriptor

```
computePass.dispatchWorkgroups(x, y, z)
```

WebGPU key concepts

- GPU
  - Discrete / integrated
  - Native GPU API (D3D12, Vulkan, Metal)
- Adapter
  - Translation layer between browser (WebGPU) and OS
- Device
  - Logical unit (multiplexing)

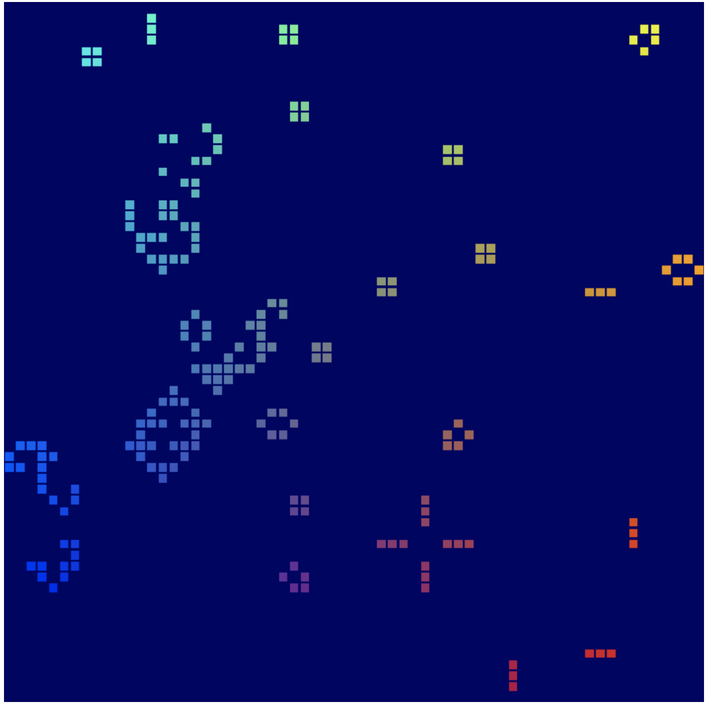


WebGPU key concepts

- Pipeline
    - Logical structure of programmable stages
- Rendering (canvas or offscreen)
    - Vertex / Fragment shaders in WGSL
- Compute
    - Any data GPGPU
- (VR ???)

Conway's game of life

- Cellular automaton, few simple rules = complex behavior



Basic WebGPU app

- Get and configure canvas context
  - Presentation format, alpha (opaque, premultiplied)
- Shader modules
  - Vertex, fragment, compute shader code
- Create resources with your data
- Create pipelines
  - Bind ground layout ~ pre-define types and purposes of GPU entities
  - Bind group ~ collection of GPU entities (buffers, textures, samplers, ...)
  - Pipeline layout ~ collection of binding groups
  - Pipeline ~ pre-define shaders and entry points
- Run rendering/compute pass

Get and configure canvas context

- getPreferredCanvasFormat() is best practice
  - select the best for device, usually `bgra8unorm` or `rgba8unorm`

```
const canvas = document.querySelector("canvas") as HTMLCanvasElement;
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const context = canvas.getContext("webgpu");

const format = navigator.gpu.getPreferredCanvasFormat();
context.configure({ device, format });
```

Shader modules ~ compute shader

```
@group(0) @binding(0) var<uniform> gridSize : vec2f;

@group(0) @binding(1) var<storage> cellStateIn : array<u32>;
@group(0) @binding(2) var<storage, read_write> cellStateOut : array<u32>;

// Conversion from 2D coordinate to 1D array index
fn cellIndex(x : u32, y : u32) -> u32 {
    // Slightly different than usual 2D -> 1D index conversion to support wrap-around effect
    return (y % u32(gridSize.y)) * u32(gridSize.x) + (x % u32(gridSize.x));
}

// Return flag whether cell is alive on specific coordinate
fn cellActive(x : u32, y : u32) -> u32 {
    return cellStateIn[cellIndex(x, y)];
}
```

Shader modules ~ compute shader

```
@compute @workgroup_size(8, 8)
fn computeMain(@builtin(global_invocation_id) cell : vec3u) {
    // Evaluate alive neighbors count
    let activeNeighbors = cellActive(cell.x + 1, cell.y + 1) + cellActive(cell.x + 1, cell.y) + ...;
    let i = cellIndex(cell.x, cell.y);

    // Conway's game of life rules
    switch activeNeighbors {
        // Active cells with 2 neighbors stay active
        case 2u : {
            cellStateOut[i] = cellStateIn[i];
        }
        // Cells with 3 neighbors become or stay active
        case 3u : {
            cellStateOut[i] = 1u;
        }
        // Cells with < 2 or > 3 neighbors become inactive
        default : {
            cellStateOut[i] = 0u;
        }
    }
}
```

Shader modules

- WebGPU compiles and links shaders for us

```
const simulationShaderModule = device.createShaderModule({
  label: "Game of life simulation compute shader",
  code: computeShader,
});
```

Create resources with your data

```
// Create, fill in and upload storage buffer for current and next state
const cellStateArray = new Uint32Array(GRID_SIZE * GRID_SIZE);
const cellStateStorage = [
  device.createBuffer({
    label: "Cell State A", size: cellStateArray.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST | GPUBufferUsage.COPY_SRC,
  }),
  device.createBuffer({
    label: "Cell State B", size: cellStateArray.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST | GPUBufferUsage.COPY_SRC,
  }),
];

// Randomize and upload initial current state
for (let i = 0; i < cellStateArray.length; i += 3) { cellStateArray[i] = Math.random() > 0.6 ? 1 : 0; }
device.queue.writeBuffer(cellStateStorage[0], 0, cellStateArray);

// Init and upload next state
for (let i = 0; i < cellStateArray.length; i++) { cellStateArray[i] = 0; }
device.queue.writeBuffer(cellStateStorage[1], 0, cellStateArray);
```

Create pipelines

```
@group(0) @binding(0) var<uniform> gridSize : vec2f;
@group(0) @binding(1) var<storage> cellStateIn : array<u32>;
@group(0) @binding(2) var<storage, read_write> cellStateOut : array<u32>;
```

```
// Define how data will be binded in shader modules
const bindGroupLayout = device.createBindGroupLayout({
  label: "Cell Bind Group Layout",
  entries: [
    {
      binding: 0, buffer: {},
      visibility: GPUShaderStage.VERTEX | GPUShaderStage.FRAGMENT | GPUShaderStage.COMPUTE,
    },
    {
      binding: 1, buffer: { type: "read-only-storage" },
      visibility: GPUShaderStage.VERTEX | GPUShaderStage.COMPUTE,
    },
    {
      binding: 2, buffer: { type: "storage" },
      visibility: GPUShaderStage.COMPUTE,
    },
```

Create pipelines

```
const bindGroups = [
  // Bind group with uniforms, cell buffer A, cell buffer B
  device.createBindGroup({
    label: "Cell renderer bind group",
    layout: bindGroupLayout,
    entries: [
      { binding: 0, resource: { buffer: uniformBuffer } },
      { binding: 1, resource: { buffer: cellStateStorage[0] } },
      { binding: 2, resource: { buffer: cellStateStorage[1] } },
    ],
  }),
  // Bind group with uniforms, cell buffer B, cell buffer A. We will be swapping between them later
  device.createBindGroup({
    label: "Cell renderer bind group",
    layout: bindGroupLayout,
    entries: [
      { binding: 0, resource: { buffer: uniformBuffer } },
      { binding: 1, resource: { buffer: cellStateStorage[1] } },
      { binding: 2, resource: { buffer: cellStateStorage[0] } },
    ],
  }),
];
```

Create pipelines

```
const pipelineLayout = device.createPipelineLayout({
  label: "Cell Pipeline Layout",
  bindGroupLayouts: [bindGroupLayout],
});

const simulationPipeline = device.createComputePipeline({
  label: "Simulation pipeline",
  layout: pipelineLayout,
  compute: {
    module: simulationShaderModule,
    entryPoint: "computeMain",
  },
});
```

Run compute pass

```
// inside render loop
// step += 1

const encoder = device.createCommandEncoder();

// Start compute pass
const computePass = encoder.beginComputePass();
computePass.setPipeline(simulationPipeline);
// Swap between state buffers
computePass.setBindGroup(0, bindGroups[step % 2]);

// Run compute pass
const workgroupCount = Math.ceil(GRID_SIZE / 8);
computePass.dispatchWorkgroups(workgroupCount, workgroupCount);

computePass.end();

// ... rendering pass

device.queue.submit([encoder.finish()]);
```

Reading data back to JS world

```
// Resource and data definition
const cellPrintState = device.createBuffer({
  label: "Cell state print buffer", size: cellStateArray.byteLength,
  usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST,
});
```

```
// Render / compute loop
encoder.copyBufferToBuffer(cellStateStorage[step % 2], 0, cellPrintState, 0, cellStateArray.byteLength);
// ...
device.queue.submit([encoder.finish()]);

// ...

await cellPrintState.mapAsync(GPUMapMode.READ, 0, cellStateArray.byteLength);
const copyArrayBuffer = cellPrintState.getMappedRange(0, cellStateArray.byteLength);
const data = new Uint32Array(copyArrayBuffer.slice(0)); // 👈
cellPrintState.unmap();
```

Resources

- Presentation & code repo @ github.com/Sorceror/PGRF2024

- webgpufundamentals.org 😍
- Tour of WGSL
- MDN WebGPU API
- WebGPU — All of the cores, none of the canvas
- WebGPU Best Practices
- WebGPU offset computer and padding visualizer
- Awesome WebGPU
- WebGPU report for current device
- WebGPU headers
- wgpu
- A trip through the Graphics Pipeline 2011