Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Skripta za vaje pri predmetih Računalniška grafika in Računalniška grafika in tehnologija iger

Žiga Lesar
2025

WebGPU

WebGPU je spletni vmesnik za dostop do grafične strojne opreme. Kot naslednik vmesnika WebGL je bolj zmogljiv, vsebuje več funkcionalnosti in omogoča pisanje bolj berljive kode brez raznovrstnih pasti, ki so še kako prisotne v WebGL. WebGPU je v zasnovi podoben sodobnim grafičnim vmesnikom za razvoj namiznih aplikacij. V primerjavi z WebGL se bistveno bolje prilagaja arhitekturi sodobnih grafičnih kartic, zato je ob spretni uporabi njegovo delovanje hitrejše in bolj predvidljivo. WebGPU poleg izrisa omogoča tudi splošno računanje preko računskih senčilnikov, kar je še posebej dobrodošlo v aplikacijah, ki se denimo zanašajo na hitro izvajanje nevronskih mrež.

Postavitev okolja

Potrebovali bomo:

- **Sodoben spletni brskalnik** (trenutno Chromium in njegovi derivati (Chrome, Opera, Edge) najbolje podpirajo standard WebGPU)
- **Urejevalnik kode** (priporočamo Sublime Text ali Visual Studio Code)
- **HTTP strežnik** (najenostavneje `python -m http.server`, sicer pa bo deloval katerikoli strežnik)

Ustvarimo direktorij `koda`.

V direktoriju `koda` ustvarimo datoteko `index.html` z vsebino

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Vaja 1</title>
    <script type="module" src="main.js"></script>
</head>
<body>
    <h1>Vaja 1</h1>
</body>
</html>
```

V direktoriju `koda` ustvarimo prazno datoteko `main.js`. V to datoteko bomo pisali JavaScript kodo za dostop do WebGPU.

V direktoriju `koda` odpremo terminal in poženemo ukaz

```
python -m http.server 3000
```

Ukaz bo pognal HTTP strežnik na lokalnem naslovu (`localhost`) na vratih 3000, kamor bo stregel vsebino direktorija, v katerem je bil ukaz pognan. Zdaj lahko spletni brskalnik usmerimo na naslov `localhost:3000` in prikaže se spletna stran z napisom **Vaja 1**. Odprimo še razvojno orodje (bližnjica v Chromiumu je F12), kjer bi morali videti konzolo brez napak.

Inicializacija vmesnika

Adapter

Najprej bomo potrebovali **adapter**, ki v grobem predstavlja fizično napravo.

```
const adapter = await navigator.gpu.requestAdapter();
```

Pri izbiri adapterja lahko po potrebi prioritiziramo nižjo porabo energije ali hitrejše delovanje, če imamo na voljo več grafičnih kartic (na primer integrirano in diskretno). V zgornjem primeru funkciji ne podamo argumenta in s tem izbiro prepustimo brskalniku.

Izbira adapterja je lahko časovno potratna operacija, zato je asinhrona. Na rezultat asinhrone operacije v JavaScriptu počakamo z operatorjem `await`.

Naprava

Po izbiri adapterja bomo zahtevali dostop do **naprave** (device), ki predstavlja glavno vstopno točko do funkcionalnosti WebGPU.

```
const device = await adapter.requestDevice();
```

Naprava je zadolžena za ustvarjanje in upravljanje z resursi. Pri izbiri naprave lahko zahtevamo uporabo razširitev ali višjih numeričnih limitov, toda s tem zmanjšamo nabor naprav, na katerih bo naša aplikacija delovala (v prvi vrsti odpadejo mobilne naprave).

Tudi izbira naprave je lahko časovno potratna operacija, zato je asinhrona.

Platno

V datoteki `index.html` izbrišimo element `<h1>` in dodajmo element `canvas` s fiksno velikostjo 512x512 pikslov:

```
<canvas width="512" height="512"></canvas>
```

Element `canvas` je platno, na katerega bo WebGPU risal geometrijo. WebGPU lahko uporabljamo tudi brez platna, če potrebujemo le njegove računske sposobnosti. Za risanje na platno moramo najprej konfigurirati povezavo med platnom in napravo. Na posamezno platno je lahko v danem trenutku povezana le ena naprava, medtem ko ena naprava lahko riše na več platen. Povezavo ustvarimo v `main.js` prek konteksta `webgpu`. Pri tem moramo določiti format barve, ki se bo uporabljal pri izrisu. Običajno je na vsaki napravi en format še posebej učinkovit. Za kateri format gre, lahko vprašamo WebGPU.

```
const canvas = document.querySelector('canvas');
const context = canvas.getContext('webgpu');
const format = navigator.gpu.getPreferredCanvasFormat();
context.configure({ device, format });
```

Brisanje platna

Vse ukaze, vključno z ukazi za upodabljanje, moramo napravi podati v razumljivi obliki. To storimo z **ukaznim kodirnikom** (command encoder), ki želene ukaze zapiše v **ukazni medpomnilnik** (command buffer), tega pa oddamo napravi v izvedbo preko **ukazne vrste** (command queue).

```
const commandEncoder = device.createCommandEncoder();
// here encode commands
const commandBuffer = commandEncoder.finish();
device.queue.submit([commandBuffer]);
```

Pred vsakim izrisom moramo platno najprej pobrisati. Brisanje platna je torej prva operacija **obhoda upodabljanja** (render pass).

```
// encode commands
const renderPass = commandEncoder.beginRenderPass({
    colorAttachments: [{
        view: context.getCurrentTexture(),
        loadOp: 'clear',
```

```
        clearValue: [0.7, 0.8, 0.9, 1],
        storeOp: 'store',
    }]
});
renderPass.end();
```

V zgornjem primeru obhod upodabljanja zaključimo takoj po začetku, brez ukazov za izris. Napravi povemo, da naj na začetku obhoda upodabljanja pobriše platno (`loadOp: 'clear'`) z določeno barvo (`clearValue: [0.7, 0.8, 0.9, 1]`), na koncu obhoda pa vse rezultate upodabljanja shrani (`storeOp: 'store'`).

V določenih primerih uporabe lahko na začetku obhoda namesto brisanja naložimo vrednosti iz prejšnjega obhoda upodabljanja (`loadOp: 'load'`), na koncu obhoda pa jih lahko zavržemo (`storeOp: 'discard'`). Te primere uporabe bomo spoznali na kasnejših vajah.

Rdeč trikotnik

Za izris trikotnika potrebujemo dva programa: prvi, **senčilnik oglišč**, določi položaj oglišč trikotnika na platnu, drugi, **senčilnik fragmentov**, pa določi barvo pikslov (natančneje fragmentov). V kakšni obliki podatki o ogliščih pridejo v obdelavo v senčilnik oglišč in v kakšni obliki so zapisane barve na izhodu senčilnika fragmentov, določa **cevovod**. Cevovod torej oba senčilnika poveže v skupno celoto in določi format podatkov na vhodu in izhodu. Zaradi učinkovitosti je pomembno, da te informacije grafični kartici podamo vnaprej.

Senčilniki

Ustvarimo datoteko `shader.wgsl`, kamor bomo pisali izvorno kodo senčilnikov v jeziku WGSL (WebGPU Shading Language).

Ustvarimo dve funkciji, `vertex` in `fragment`:

```
fn vertex() -> vec4f {
    return vec4f(0, 0, 0, 1);
}

fn fragment() -> vec4f {
    return vec4f(1, 0, 0, 1);
}
```

Zgornja koda se bo sicer prevedla, toda za uporabo funkcij v vlogi senčilnika oglišč in senčilnika fragmentov moramo upoštevati nekaj osnovnih pravil:

- Senčilnik oglišč mora biti dekoriran z dekoratorjem `@vertex`
- Senčilnik fragmentov mora biti dekoriran z dekoratorjem `@fragment`
- Senčilnik oglišč mora položaj posameznega oglišča zapisati v vgrajeno spremenljivko `position`
- Senčilnik fragmentov mora barvo posameznega fragmenta zapisati v izhodno sliko, določeno z indeksom (seznam izhodnih slik bomo podali pri konfiguraciji cevovoda)

Z dodanimi dekoratorji bo koda senčilnikov izgledala tako:

```
@vertex
fn vertex() -> @builtin(position) vec4f {
    return vec4f(0, 0, 0, 1);
}

@fragment
fn fragment() -> @location(0) vec4f {
    return vec4f(1, 0, 0, 1);
}
```

V datoteki `main.js` moramo kodo senčilnikov najprej pridobiti s strežnika, za kar uporabimo asinhrono funkcijo `fetch`, iz odgovora strežnika pa izluščimo vsebino v obliki besedila:

```
const code = await fetch('shader.wgsl').then(response => response.text());
```

Senčilnike prevedemo s klicem funkcije `createShaderModule`:

```
const module = device.createShaderModule({ code });
```

Cevovod

Senčilnike in obliko vhodnih in izhodnih podatkov določimo v **cevovodu**. Ustvarjanje cevovoda zahteva veliko dela od gonilnika, saj mora celotno konfiguracijo temeljito validirati, da med izvajanjem ne pride do raznovrstnih napak. Cevovode zato vedno ustvarimo vnaprej in ne v zadnjem trenutku pred prvo uporabo. Obstajata dve vrsti cevovoda, grafični in računski. Na vajah bomo uporabljali le grafičnega.

```
const pipeline = device.createRenderPipeline({
    vertex: {
        module,
    },
    fragment: {
        module,
        targets: [{ format }],
    },
    layout: 'auto',
});
```

V zgornjem primeru določimo senčilniški modul za obe stopnji cevovoda. Pri senčnilniku fragmentov moramo določiti tudi format izhodnih slik. Indeks 0 v dekoratorju `@location(0)` se nanaša na ta seznam. Zadnji podatek pri ustvarjanju cevovoda je razpored cevovoda (pipeline layout), ki ga bomo obravnavali na kasnejših vajah. Do nadaljnjega bomo ustvarjanje razporeda cevovoda prepustili gonilniku, ki razpored izlušči iz same kode senčilnikov, toda to lahko privede do raznovrstnih težav, zato je v praksi priporočljivo ročno ustvarjanje razporeda.

Obhod upodabljanja

V obhodu upodabljanja moramo določiti cevovod in izstaviti klic izrisa (draw call):

```
renderPass.setPipeline(pipeline);
renderPass.draw(3);
```

V zgornjem izrisu smo določili tri oglišča, torej bomo izrisali en trikotnik.

Če kodo poženemo, bomo videli prazno platno. Senčilnik oglišč bo namreč vsem ogliščem določil enak položaj na platnu. Na nekakšen način moramo razlikovati med oglišči in jim določiti različne položaje. Najenostavneje to dosežemo prek vgrajene spremenljivke `vertex_index`, ki bo ogliščem našega trikotnika priredil indekse 0, 1 in 2. Spremenljivko `vertex_index` lahko v kodo vključimo kot parameter funkcije `vertex`:

```
@vertex
fn vertex(@builtin(vertex_index) vertexIndex) -> @builtin(position) vec4f {
    if (vertexIndex == 0) {
        return vec4f(0, 0, 0, 1);
    } else if ( ... ) {
        ...
    }
}
```

Kodo lahko še malenkost poenostavimo z uporabo seznamov:

```
const positions = array(
    vec2f(-0.5, -0.5),
    vec2f( 0.5, -0.5),
    vec2f( 0.0,  0.5),
);

@vertex
fn vertex(@builtin(vertex_index) vertexIndex) -> @builtin(position) vec4f {
    return vec4f(positions[vertexIndex], 0, 1);
}
```

Posodobljena koda bi morala na platnu izrisati rdeč trikotnik.

Obarvan trikotnik

Trikotnik bi želeli obarvati tako, da obarvamo vsako oglišče s svojo barvo, v notranjosti trikotnika pa bi se barve prelivale. Barve lahko ogliščem določimo na podoben način kot položaje:

```
const colors = array(
    vec4f(1, 0, 0, 1),
    vec4f(0, 1, 0, 1),
    vec4f(0, 0, 1, 1),
);
```

Za prelivanje barv v notranjosti trikotnika bomo uporabili eno izmed osnovnih operacij grafične kartice, **interpolacijo**. Ker je interpolacija tako pogosta operacija v računalniški grafiki, je zaradi višje učinkovitosti na grafičnih karticah običajno implementirana kar v strojni opremi. Mi moramo le določiti spremeljivke, katerih vrednosti želimo interpolirati. Te spremenljivke imenujemo **interpoliranke**. Njihove vrednosti določimo v senčilniku oglišč, grafična kartica pa jih interpolira med izrisom na platno. Interpolirane vrednosti nato prevzame senčilnik fragmentov, ki jih lahko uporabi denimo pri izračunu barve fragmenta.

Ker funkcije v naših senčilnikih ne morejo vrniti več vrednosti, bomo morali strukturo programa nekoliko spremeniti. Določili bomo štiri strukture, ki bodo definirale vhode in izhode senčilnikov:

```
struct VertexInput {
    @builtin(vertex_index) vertexIndex: u32,
```

```
}

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(0) color: vec4f, // value before interpolation
}

struct FragmentInput {
    @location(0) color: vec4f, // value after interpolation
}

struct FragmentOutput {
    @location(0) color: vec4f, // note, the output location is in no way relat
}
```

Posodobimo še senčilnika, da bosta uporabljala zgornje strukture:

```
@vertex
fn vertex(input: VertexInput) -> VertexOutput {
    var output: VertexOutput;

    output.position = vec4f(positions[input.vertexIndex], 0, 1);
    output.color = colors[input.vertexIndex];

    return output;
}

@fragment
fn fragment(input: FragmentInput) -> FragmentOutput {
    var output: FragmentOutput;

    output.color = input.color;

    return output;
}
```

Tako posodobljena koda bi morala brez sprememb v datoteki `main.js` izrisati trikotnik, v katerem se barve gladko prelivajo med rdečo, zeleno in modro.

Naloge

1. Podatke o ogliščih spremeni in dopolni, tako da program izriše kvadrat. Po potrebi posodobi tudi obhod upodabljanja.

2. Spremeni koordinate oglišč, tako da se bo kvadrat dotikal robov platna. Katere vrednosti x in y dosežejo tako stanje?
3. Spreminjaj koordinato z, dokler kvadrat ne izgine. Pri katerih vrednostih je kvadrat prikazan na platnu?
4. Spreminjaj koordinato w. Kaj se dogaja s kvadratom in zakaj?
5. Z rotacijsko matriko zavrti kvadrat za kot 20 stopinj. Pomagaj si s funkcijami mat4x4 (oz. mat4x4f), cos in sin.

Ločeni atributi oglišč

V prejšnji vaji smo podatke o ogliščih (položaje in barve) zapisali v senčilnik. Seveda tak pristop ni skalabilen. Podatke bomo zato zapisali v grafični pomnilnik in jih v senčilnik poslali v obhodu upodabljanja.

Senčilnik

Najprej v senčilniku oglišč dopolnimo `VertexInput` z dvema novima spremenljivkama:

```
struct VertexInput {
    @location(0) position: vec2f,
    @location(1) color: vec4f,
}
```

Posodobiti moramo tudi funkcijo `vertex`:

```
output.position = vec4(input.position, 0, 1);
output.color = input.color;
```

Spremenljivke, ki opisujejo posamezno oglišče, imenujemo **atributi**. Naš senčilnik bo torej sprejel oglišče z dvema atributoma: položajem na lokaciji 0 in barvo na lokaciji 1. Te številke bomo potrebovali pri ustvarjanju cevovoda.

V zgornji kodi opazimo, da smo izbrisali vhodno spremenljivko `vertexIndex`. Prav tako lahko izbrišemo seznama položajev in barv, saj jih bomo prestavili v `main.js`. Senčilnik se je s tem bistveno poenostavil.

Medpomnilniki

Najprej podatke o položajih zapišimo v glavni pomnilnik v `main.js`:

```
const positions = new Float32Array([
    -0.5, -0.5,
     0.5, -0.5,
     0.0,  0.5,
]);
```

Vidimo, da iz kode ni razvidno, da gre za dvodimenzionalne vektorje. Ta informacija je del formata podatkov, kar bomo napravi sporočili pri ustvarjanju cevovoda.

V naslednjem koraku ustvarimo **medpomnilnik** (buffer), ki predstavlja posamezno alokacijo grafičnega pomnilnika:

```
const positionBuffer = device.createBuffer({
    size: positions.byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
```

Pri ustvarjanju medpomnilnika moramo napravi sporočiti njegovo velikost in uporabo. Zastavica `VERTEX` napravi sporoča, da bodo podatki iz medpomnilnika uporabljeni v senčilniku oglišč, za zapisovanje v medpomnilnik pa je dodatno potrebna zastavica `COPY_DST`.

Ukaz za zapisovanje podatkov v medpomnilnik izstavimo v ukazno vrsto:

```
device.queue.writeBuffer(positionBuffer, 0, positions);
```

V zgornji vrstici število 0 predstavlja odmik v številu bajtov od začetka medpomnilnika, kjer bomo začeli pisanje.

Na enak način pripravimo podatke o barvah:

```
const colors = new Float32Array([
    1, 0, 0, 1,
    0, 1, 0, 1,
    0, 0, 1, 1,
]);

const colorBuffer = device.createBuffer({
    size: colors.byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});

device.queue.writeBuffer(colorBuffer, 0, colors);
```

Cevovod

Zdaj se lahko lotimo cevovoda. Pri tem moramo vnaprej sporočiti napravi, kakšne medpomnilnike bomo uporabljali, kakšen bo format atributov in njihov razpored znotraj medpomnilnikov ter na katere lokacije v senčilniku bodo podatki vezani.

Najprej ustvarimo opis prvega medpomnilnika, ki vsebuje položaje oglišč:

```
const positionBufferLayout = {
    arrayStride: 8,
    attributes: [{
        shaderLocation: 0,
        offset: 0,
        format: 'float32x2',
    }]
};
```

Z zgornjim opisom napravi sporočimo, da je v tem medpomnilniku le en atribut, ki je vezan na lokacijo 0 v senčilniku oglišč, njegovi podatki pa so v medpomnilniku zapisani kot dvodimenzionalni vektorji 32-bitnih števil s plavajočo vejico. Položaj prvega oglišča je od začetka medpomnilnika zamaknjen za 0 bajtov, položaj vsakega naslednjega oglišča pa je zamaknjen za nadaljnjih 8 bajtov.

Podoben opis potrebujemo za drugi medpomnilnik, ki vsebuje barve oglišč:

```
const colorBufferLayout = {
    arrayStride: 16,
    attributes: [{
        shaderLocation: 1,
        offset: 0,
        format: 'float32x4',
    }]
};
```

Gre torej za medpomnilnik, ki vsebuje en atribut, vezan na lokacijo 1 v senčilniku oglišč. Njegovi podatki so v medpomnilniku zapisani kot štiridimenzionalni vektorji 32-bitnih števil s plavajočo vejico. Barva prvega oglišča je od začetka medpomnilnika zamaknjena za 0 bajtov, barva vsakega naslednjega oglišča pa za nadaljnjih 16 bajtov.

Zgornja opisa medpomnilnikov napravi podamo ob ustvarjanju cevovoda:

```
const pipeline = device.createRenderPipeline({
    vertex: {
        module,
        buffers: [positionBufferLayout, colorBufferLayout],
    },
    fragment: {
        module,
        targets: [{ format }],
    },
    layout: 'auto',
});
```

Obhod upodabljanja

V obhodu upodabljanja moramo po klicu `setPipeline` le še povezati medpomnilnika s cevovodom:

```
renderPass.setVertexBuffer(0, positionBuffer);
renderPass.setVertexBuffer(1, colorBuffer);
```

Številke v zgornji kodi se nanašajo na seznam medpomnilnikov v opisu cevovoda, ne na lokacije atributov v senčilniku.

Tako popravljena koda bi se morala izvesti brez napak in na platno izrisati enak trikotnik.

Prepleteni atributi oglišč

V zgornjem primeru smo podatke za posamezno oglišče pridobili iz dveh ločenih medpomnilnikov. Pri večjem številu oglišč je tak dostop do pomnilnika zelo potraten zaradi slabega izkoristka predpomnilnika grafične kartice. Boljšo učinkovitost dosežemo s prepletanjem atributov, tako da so atributi posameznega oglišča tudi v pomnilniku blizu skupaj.

V `main.js` bomo podatke o ogliščih zapisali v en seznam, imenovan `vertices`:

```
const vertices = new Float32Array([
    // position    // color
    -0.5, -0.5,    1, 0, 0, 1,
     0.5, -0.5,    0, 1, 0, 1,
     0.0,  0.5,    0, 0, 1, 1,
]);
```

Podatke nato zapišemo v medpomnilnik `vertexBuffer`:

```
const vertexBuffer = device.createBuffer({
    size: vertices.byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});

device.queue.writeBuffer(vertexBuffer, 0, vertices);
```

Ustvarimo primeren opis medpomnilnika:

```
const vertexBufferLayout = {
    arrayStride: 24,
    attributes: [
        {
            shaderLocation: 0,
            offset: 0,
            format: 'float32x2',
        },
        {
            shaderLocation: 1,
            offset: 8,
            format: 'float32x4',
        },
    ]
};
```

Tokrat sta v medpomnilniku shranjena dva atributa z različnima odmikoma od začetka medpomnilnika in različnima formatoma. Velikost oglišča se je povečala na 24 bajtov.

Posodobimo še konfiguracijo senčilnika oglišč v cevovodu:

```
buffers: [vertexBufferLayout],
```

Temu primerno spremenimo še obhod upodabljanja:

```
renderPass.setVertexBuffer(0, vertexBuffer);
```

Vse sledove ločenih medpomnilnikov lahko zdaj izbrišemo.

Indeksiranje

Še zadnja optimizacija bo indeksirano upodabljanje, ki nam omogoča ponovno uporabo že definiranih oglišč. Za demonstracijo koncepta bomo namesto trikotnika izrisali kvadrat. Ker je kvadrat sestavljen iz 2 trikotnikov, potrebujemo 6 oglišč, od katerih sta 2 podvojeni:

```
const vertices = new Float32Array([
    // 1st triangle
    -0.5, -0.5,    1, 0, 0, 1,
     0.5, -0.5,    0, 1, 0, 1,
    -0.5,  0.5,    0, 0, 1, 1,
```

```
    // 2nd triangle
    -0.5,  0.5,    0, 0, 1, 1,
     0.5, -0.5,    0, 1, 0, 1,
     0.5,  0.5,    1, 1, 0, 1,
]);
```

V obhodu upodabljanja izrišemo 6 oglišč:

```
renderPass.draw(6);
```

Le s tema dvema popravkoma bi morala koda izrisati kvadrat.

Podvajanju oglišč se lahko izognemo tako, da najprej vsako oglišče v medpomnilniku oglišč definiramo le enkrat:

```
const vertices = new Float32Array([
    // position    // color
    -0.5, -0.5,    1, 0, 0, 1,
     0.5, -0.5,    0, 1, 0, 1,
    -0.5,  0.5,    0, 0, 1, 1,
     0.5,  0.5,    1, 1, 0, 1,
]);
```

Nato zgradimo še **medpomnilnik indeksov**:

```
const indices = new Uint32Array([
    // 1st triangle
    0, 1, 2,
    // 2nd triangle
    2, 1, 3,
]);

const indexBuffer = device.createBuffer({
    size: indices.byteLength,
    usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});

device.queue.writeBuffer(indexBuffer, 0, indices);
```

Pri tem smo uporabili 32-bitna nepredznačena cela števila (`Uint32Array`) in napravi sporočili, da bo medpomnilnik uporabljen kot medpomnilnik indeksov (`INDEX`).

Posodobiti moramo še obhod upodabljanja, tako da nastavimo medpomnilnik indeksov z določenim podatkovnim tipom (`uint32`) in namesto funkcije `draw` kličemo funkcijo

`drawIndexed`, ki ji podamo število indeksov:

```
renderPass.setIndexBuffer(indexBuffer, 'uint32');
renderPass.drawIndexed(indices.length);
```

Na platnu bi morali videti obarvan kvadrat.

V opisanem primeru gre za nepotrebno optimizacijo, saj je količina podatkov zelo majhna. Praktični 3D modeli pa posamezno oglišče lahko uporabijo tudi v 6 ali več trikotnikih, zato je taka optimizacija kritičnega pomena. Vsi modeli, ki jih bomo uporabljali na vajah, bodo indeksirani.

Animacija

Če želimo kvadrat programsko animirati, moramo najti način za prenos podatkov o njegovi transformaciji v senčilnik. Podatki o transformaciji so skupni vsem ogliščem in vsem fragmentom, zato v senčilniku delujejo podobno kot konstante. Imenujemo jih **uniforme**. V nasprotju s pravimi konstantami lahko uniforme med zaporednimi klici izrisa spreminjamo in s tem ustvarimo animacijo.

Uniforme in drugi zunanji viri so organizirani v **skupine**, znotraj skupine pa ima vsak vir svojo **številko vezave**. Vse vire v posamezni skupini v senčilnik povežemo z enim samim funkcijskim klicem.

Senčilnik

Denimo, da želimo kvadratu dodati translacijo, ki jo predstavimo z 2D vektorjem. Najprej jo dodamo v senčilnik oglišč:

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

Translacijo prištejemo položaju oglišča:

```
output.position = vec4(input.position + translation, 0, 1);
```

Translacijo smo v zgornji kodi dodelili skupini 0 in ji določili številko vezave 0. Te številke bodo pomembne pri ustvarjanju skupine vezav.

Medpomnilnik

Vrednosti uniform senčilnik pridobi iz medpomnilnika. Ustvarimo medpomnilnik, ki naj bo vsaj tako velik, da lahko hrani dve števili s plavajočo vejico:

```
const uniformBuffer = device.createBuffer({
    size: 8,
    usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
```

Pri tem smo napravi sporočili, da bo medpomnilnik uporabljen kot uniforma.

Skupina vezav

Za vezavo medpomnilnika v senčilnik moramo ustvariti še skupino vezav:

```
const bindGroup = device.createBindGroup({
    layout: pipeline.getBindGroupLayout(0),
    entries: [
        { binding: 0, resource: uniformBuffer },
    ]
});
```

Razpored uniform v posamezni skupini in njihove tipe določa **razpored skupine** (bind group layout), razpored vseh skupin v celotnem cevovodu pa določa **razpored cevovoda**. Stvarjenje slednjega smo v prvi vaji prepustili gonilniku (`layout: 'auto'`), tako da lahko razpored skupine pridobimo kar prek funkcije `pipeline.getBindGroupLayout`, ki ji podamo številko skupine. Tako ustvarjena skupina vezav bo posledično veljavna le v tem cevovodu.

Nazadnje v obhodu upodabljanja skupino povežemo s senčilnikom:

```
renderPass.setBindGroup(0, bindGroup);
```

Animacija

Do tega trenutka je naša aplikacija izris izvedla le enkrat, za animacijo pa bo treba napisati zanko, ki bo najprej posodobila transformacijo kvadrata, nato pa izrisala posodobljeno sceno. Potrebujemo torej dve funkciji, `update` za posodabljanje in `render` za izris, ki ju bomo klicali večkrat (običajno 60-krat) na sekundo:

```
function update() {
    // user input, animations, AI ...
}

function render() {
    // clear the canvas, render
}

function frame() {
    update();
    render();
    requestAnimationFrame(frame);
}

requestAnimationFrame(frame);
```

Za implementacijo zanke smo uporabili v brskalnik vgrajeno funkcijo `requestAnimationFrame`, ki podano funkcijo (`frame`) izvede pred naslednjim osveževanjem zaslona.

Če želimo, da se kvadrat premika po krožnici, lahko animacijo napišemo tako:

```
function update() {
    const time = performance.now() / 1000;
    const radius = 0.5;
    const frequency = 0.5;
    const x = radius * Math.cos(frequency * time * 2 * Math.PI);
    const y = radius * Math.sin(frequency * time * 2 * Math.PI);

    device.queue.writeBuffer(uniformBuffer, 0, new Float32Array([x, y]));
}
```

Transformacije z matrikami

Transformacije lahko posplošimo z uporabo matrik. V računalniški grafiki se najpogosteje uporabljajo realne matrike velikosti 4x4, saj lahko z njimi predstavimo poljubne afine in celo perspektivne transformacije. Poleg tega lahko več transformacij združimo v eno samo matriko in s tem še pospešimo izračun.

Najprej v senčilniku zamenjajmo spremenljivko `translation` tipa `vec2f` z matriko `matrix` tipa `mat4x4f`:

```
@group(0) @binding(0) var<uniform> matrix: mat4x4f;
```

Matriko pomnožimo s položajem oglišča:

```
output.position = matrix * vec4(input.position, 0, 1);
```

Zdaj lahko translacijo v funkciji `update` nadomestimo z matriko:

```
device.queue.writeBuffer(uniformBuffer, 0, new Float32Array([
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    x, y, 0, 1,
]);
```

Ne pozabimo še povečati velikosti medpomnilnika, ki mora biti zdaj velik vsaj 64 bajtov.

Matrike so na grafični kartici zapisane po stolpcih, zato matrika v zgornjem zapisu izgleda transponirana.

Naloge

1. Napiši funkcijo, ki ustvari model kroga. Model naj bo indeksiran in sestavljen iz trikotnikov, ki se stikajo v središču kroga. Število oglišč na obodu kroga naj bo podano kot parameter funkcije. Barva vseh oglišč je lahko enaka.
2. Napiši funkcijo, ki ustvari regularno mrežo. Ločljivost mreže v smeri x in y naj bo podana s parametroma funkcije. Model naj bo indeksiran. Koliko grafičnega pomnilnika porabi indeksiran model? Kaj pa neindeksiran?
3. Kodo posodobi tako, da bo izrisala vse modele, podane s seznamom. Vsak model naj ima svoj medpomnilnik oglišč, medpomnilnik indeksov, medpomnilnik uniform in skupino vezave.

Kocka

V tej vaji bomo kvadrat nadgradili v kocko. V ta namen bomo posodobili podatke, cevovod in senčilnik. Poskrbeli bomo še za postavitev kamere v sceno in vse potrebne transformacije.

Podatki

Oglišča kocke bomo postavili na koordinate ±1. Položajem bomo dodali še homogeno koordinato, tako da jih bomo lažje obdelovali v senčilniku. Vsako oglišče bo imelo tudi svojo barvo, ki bo odražala položaj oglišča. Kocka ima 6 kvadratnih ploskev, vsaka je predstavljena z 2 trikotnikoma, kar skupaj znese 36 indeksov.

```
const vertices = new Float32Array([
    // positions         // colors          // index
    -1, -1, -1,  1,      0,  0,  0,  1,     //   0
    -1, -1,  1,  1,      0,  0,  1,  1,     //   1
    -1,  1, -1,  1,      0,  1,  0,  1,     //   2
    -1,  1,  1,  1,      0,  1,  1,  1,     //   3
     1, -1, -1,  1,      1,  0,  0,  1,     //   4
     1, -1,  1,  1,      1,  0,  1,  1,     //   5
     1,  1, -1,  1,      1,  1,  0,  1,     //   6
     1,  1,  1,  1,      1,  1,  1,  1,     //   7
]);

const indices = new Uint32Array([
    0, 1, 2,    2, 1, 3,
    4, 0, 6,    6, 0, 2,
    5, 4, 7,    7, 4, 6,
    1, 5, 3,    3, 5, 7,
    6, 2, 7,    7, 2, 3,
    1, 0, 5,    5, 0, 4,
]);
```

Format podatkov

Nova oglišča so velika 32 bajtov, pri čemer prvih 16 bajtov zaseda položaj, drugih 16 bajtov pa barva oglišča.

```
const vertexBufferLayout = {
    arrayStride: 32,
    attributes: [
        {
```

```
            shaderLocation: 0,
            offset: 0,
            format: 'float32x4',
        },
        {
            shaderLocation: 1,
            offset: 16,
            format: 'float32x4',
        },
    ],
};
```

Senčilnik

Senčilnik posodobimo tako, da bo atribut `position` odražal zgornje spremembe:

```
@location(0) position: vec4f,
```

V glavni funkciji položaju ni več treba dodati dveh fiksnih komponent, saj jih v senčilnik prinesemo že z atributom:

```
output.position = matrix * input.position;
```

Senčilnik in podatki so zdaj pripravljeni za delo v treh dimenzijah.

Transformacije

Knjižnica glMatrix

Transformacije 3D modelov bomo izvajali s pomočjo 4x4 matrik. Pisanje lastnih funkcij za delo z matrikami in vektorji je dobra vaja za programiranje, mi pa bomo uporabili kar obstoječo knjižnico `glMatrix`, ki je optimizirana za čim hitrejše izvajanje.

Knjižnico dobimo na naslovu https://raw.githubusercontent.com/UL-FRI-LGM/webgpu-examples/master/lib/glm.js.

V skripto `main.js` uvozimo kodo za delo s 4x4 matrikami:

```
import { mat4 } from './glm.js';
```

Postavitev scene

Potrebujemo tri matrike: transformacijsko matriko modela, ki model iz lokalnega prostora postavi v globalni prostor, transformacijsko matriko pogleda, ki model iz globalnega prostora postavi v gledišči prostor, in projekcijsko matriko, ki model iz glediščnega prostora postavi v rezalni prostor. Grafična kartica bo po rezanju izvedla še perspektivno deljenje, rezultat katerega bodo točke v normaliziranem prostoru zaslona. Temu sledita le še zaslonska preslikava in rasterizacija.

Glede koordinatnih sistemov bomo sledili zgledu, ki ga postavljajo vsi večji grafični pogoni: koordinatni sistemi bodo desnosučni, pogled kamere je usmerjen vzdolž negativne smeri lokalne osi *z*.

Matrika pogleda in projekcijska matrika naj bosta za zdaj fiksni, matriko modela pa bomo spreminjali v funkciji `update`. Kocko bomo postavili v izhodišče, kamero pa bomo premaknili za 5 enot nazaj, tako da bo vidna celotna kocka. Najprej ustvarimo matriko pogleda, ki je inverzna transformacijski matriki kamere:

```
const viewMatrix = mat4.fromTranslation(mat4.create(), [0, 0, -5]);
```

Tudi projekcijska matrika naj bo fiksna, z vertikalnim zornim kotom 1 radian, razmerjem med višino in širino zaslona 1 (platno je namreč kvadratno), ter sprednjo in zadnjo rezalno ravnino na razdalji 0.01 in 1000 enot:

```
const projectionMatrix = mat4.perspectiveZO(mat4.create(), 1, 1, 0.01, 1000);
```

Ustvarimo še matriko modela:

```
const modelMatrix = mat4.create();
```

Kocko animiramo v funkciji `update`:

```
const time = performance.now() / 1000;
modelMatrix.identity().rotateX(time * 0.6).rotateY(time * 0.7);
```

Zgornja koda matriko modela najprej ponastavi, sicer bi se ohranjala skozi zaporedne slike animacije.

Prenos transformacije v senčilnik

Senčilnik sprejema le eno matriko, ki predstavlja združeno transformacijo modela, pogleda in projekcije. V funkciji `render` jih zmnožimo in pri tem pazimo na vrstni red množenja:

```
const matrix = mat4.create()
    .multiply(projectionMatrix)
    .multiply(viewMatrix)
    .multiply(modelMatrix);
```

Rezultat zapišemo v medpomnilnik:

```
device.queue.writeBuffer(uniformBuffer, 0, matrix);
```

Na platnu bi morala biti vidna vrteča se kocka.

Globinska slika

Nenavadno prekrivanje ploskev kocke je posledica odsotnosti globinskega testa. Za pravilno delovanje moramo ustvariti globinsko sliko, vključiti globinski test in globinsko sliko počistiti pred vsakim izrisom.

Ustvarimo globinsko sliko v velikosti platna in s primernim globinskim formatom:

```
const depthTexture = device.createTexture({
    size: [canvas.width, canvas.height],
    format: 'depth24plus',
    usage: GPUTextureUsage.RENDER_ATTACHMENT,
});
```

Vključimo globinski test v cevovodu:

```
depthStencil: {
    depthWriteEnabled: true,
    depthCompare: 'less',
    format: 'depth24plus',
}
```

Globinski sliko pripnemo na cevovod v obhodu upodabljanja:

```
depthStencilAttachment: {
    view: depthTexture,
    depthClearValue: 1,
    depthLoadOp: 'clear',
    depthStoreOp: 'discard',
}
```

S tem bi moralo nepravilno prekrivanje ploskev izginiti.

Komponentni sistem

Trenutno je v datoteki `main.js` združeno veliko funkcionalnosti, ki bi jo bilo bolje ločiti na več delov, ki jih bomo lahko ponovno uporabili. Refaktorizacijo bomo pričeli pri transformacijah. Ustvarili bomo dva razreda, `Transform` in `Camera`, ki bosta zadolžena za ustvarjanje transformacijskih matrik preko intuitivnih parametrov. Nato bomo ustvarili še razred `Node`, ki bo predstavljal posamezno vozlišče v grafu scene in vseboval seznam pripetih komponent.

Komponenti Transform in Camera

Začnimo z razredom `Transform`, ki ga zapišimo v datoteko `Transform.js`:

```
import { mat4 } from './glm.js';

export class Transform {

    constructor({
        rotation = [0, 0, 0, 1],
        translation = [0, 0, 0],
        scale = [1, 1, 1],
    } = {}) {
        this.rotation = rotation;
        this.translation = translation;
        this.scale = scale;
    }

    get matrix() {
        return mat4.fromRotationTranslationScale(mat4.create(),
            this.rotation, this.translation, this.scale);
    }

}
```

Razred je napisan tako, da ga lahko enostavno instanciramo in pri tem opcijsko dodamo parametre vrtenja, premika in raztega. Vrtenje je, kot v večini sodobnih grafičnih pogonov, predstavljeno s kvaternionom.

Dodajmo še razred `Camera` za predstavitev perspektivne kamere in ga zapišimo v datoteko `Camera.js`:

```
import { mat4 } from './glm.js';

export class Camera {

    constructor({
        aspect = 1,
        fovy = 1,
        near = 0.01,
        far = 1000,
    } = {}) {
        this.aspect = aspect;
        this.fovy = fovy;
        this.near = near;
        this.far = far;
    }

    get matrix() {
        const { fovy, aspect, near, far } = this;
        return mat4.perspectiveZO(mat4.create(), fovy, aspect, near, far);
    }

}
```

Graf scene

Ustvarjeni komponenti bomo pripeli na objekte v sceni, ki jih bomo predstavili z grafom. Bolj natančno, ustvarili bomo razred `Node`, ki bo predstavljal objekte v grafu scene. Vsak objekt ima lahko več otrok in kvečjemu enega starša.

Razred `Node` zapišimo v datoteko `Node.js`.

```
export class Node {

    constructor() {
        this.parent = null;
        this.children = [];
        this.components = [];
    }
```

```
    addChild(node) {
        node.parent?.removeChild(node);
        node.parent = this;
        this.children.push(node);
    }

    removeChild(node) {
        const index = this.children.indexOf(node);
        if (index >= 0) {
            this.children.splice(index, 1);
            node.parent = null;
        }
    }

    traverse(before, after) {
        before?.(this);
        for (const child of this.children) {
            child.traverse(before, after);
        }
        after?.(this);
    }

    linearize() {
        const array = [];
        this.traverse(node => array.push(node));
        return array;
    }

    filter(predicate) {
        return this.linearize().filter(predicate);
    }

    find(predicate) {
        return this.linearize().find(predicate);
    }

    map(transform) {
        return this.linearize().map(transform);
    }

    addComponent(component) {
        this.components.push(component);
    }

    removeComponent(component) {
        this.components = this.components.filter(c => c !== component);
    }
```

```
    removeComponentsOfType(type) {
        this.components = this.components.filter(component => !(components inst
    }

    getComponentOfType(type) {
        return this.components.find(component => component instanceof type);
    }

    getComponentsOfType(type) {
        return this.components.filter(component => component instanceof type);
    }

}
```

Razred `Node` vsebuje osnovne metode za delo z grafom scene in za upravljanje s komponentami.

Če želimo v sceno dodati objekt z določeno transformacijo, lahko to zdaj enostavno storimo z uporabo zgornjih razredov:

```
const object = new Node();
object.addComponent(new Transform({
    translation: [1, 2, 3]
}));

const scene = new Node();
scene.addChild(object);
```

Ker lahko na posamezen objekt dodamo poljubno količino komponent (vključujoč transformacije), je pridobivanje transformacijskih matrik nekoliko bolj težavno. Poleg tega moramo upoštevati še celoten graf scene in s tem povezano združevanje transformacijskih matrik. Za enostavnejše delo s transformacijami ustvarimo še datoteko `SceneUtils.js`, kamor bomo zapisali funkcije za združevanje matrik:

```
import { mat4 } from './glm.js';

import { Transform } from './Transform.js';
import { Camera } from './Camera.js';

export function getLocalModelMatrix(node) {
    const matrix = mat4.create();
    for (const transform of node.getComponentsOfType(Transform)) {
        matrix.multiply(transform.matrix);
    }
    return matrix;
```

```
}

export function getGlobalModelMatrix(node) {
    if (node.parent) {
        const parentMatrix = getGlobalModelMatrix(node.parent);
        const modelMatrix = getLocalModelMatrix(node);
        return parentMatrix.multiply(modelMatrix);
    } else {
        return getLocalModelMatrix(node);
    }
}

export function getLocalViewMatrix(node) {
    return getLocalModelMatrix(node).invert();
}

export function getGlobalViewMatrix(node) {
    return getGlobalModelMatrix(node).invert();
}

export function getProjectionMatrix(node) {
    return node.getComponentOfType(Camera)?.matrix ?? mat4.create();
}
```

Uporaba komponentnega sistema

Zdaj lahko veliko funkcionalnosti naše aplikacije poenostavimo z uporabo komponentnega sistema. Najprej v datoteko `main.js` uvozimo vse potrebne razrede in funkcije:

```
import { quat, mat4 } from './glm.js';
import { Transform } from './Transform.js';
import { Camera } from './Camera.js';
import { Node } from './Node.js';
import {
    getGlobalModelMatrix,
    getGlobalViewMatrix,
    getProjectionMatrix,
} from './SceneUtils.js';
```

Nato ustvarimo sceno:

```
const model = new Node();
model.addComponent(new Transform());

const camera = new Node();
```

```
camera.addComponent(new Camera());
camera.addComponent(new Transform({
    translation: [0, 0, 5]
}));

const scene = new Node();
scene.addChild(model);
scene.addChild(camera);
```

Posamezne transformacijske matrike lahko izbrišemo.

V funkciji `render` poenostavimo pridobivanje transformacijskih matrik:

```
const modelMatrix = getGlobalModelMatrix(model);
const viewMatrix = getGlobalViewMatrix(camera);
const projectionMatrix = getProjectionMatrix(camera);
```

V funkciji `update` lahko posodobimo vse komponente vseh objektov v sceni hkrati, tako da pokličemo njihove lastne funkcije `update`, če so na voljo:

```
scene.traverse(node => {
    for (const component of node.components) {
        component.update?.();
    }
});
```

S tem lahko animacijo kocke izločimo iz glavne funkcije `update` in jo dodamo kot komponento:

```
model.addComponent({
    update() {
        const time = performance.now() / 1000;
        const transform = model.getComponentOfType(Transform);
        const rotation = transform.rotation;

        quat.identity(rotation);
        quat.rotateX(rotation, rotation, time * 0.6);
        quat.rotateY(rotation, rotation, time * 0.7);
    }
});
```

Naloge

1. Napiši komponento, ki objekt zvezno premika med dvema točkama v prostoru.
2. Napiši komponento `Model`, ki naj v konstruktorju prejme seznam oglišč in seznam indeksov. Program naj se pred izrisom sprehodi po grafu scene in prenese podatke modelov na grafično kartico. Za vsak model ustvari medpomnilnik uniform za hranjenje transformacije ter pripadajočo skupino vezav. Tako pripravljene podatke uporabi pri izrisu grafa scene.

Teksturiranje

V tej vaji bomo na 3D model nalepili 2D **teksturo**. Teksture so lahko različnih razsežnosti in formatov ter lahko vsebujejo več nivojev piramide slik (mipmap level). Podatki v teksturi so zapisani s **teksli**, katerih vrednosti **vzorčimo** (sample) z **vzorčevalnikom** (sampler) na podanih **teksturnih koordinatah** (texture coordinates, UV coordinates), ki določajo točko v **teksturnem prostoru** (texture space).

Prenos s strežnika

V prvem koraku izberemo sliko in jo shranimo v direktorij aplikacije pod imenom `image.png`. Slika naj ne bo prevelika, tako v smislu ločljivosti kot velikosti datoteke. Za večino primerov uporabe bo dovolj slika ločljivosti 512x512 pikslov. Ločljivost je lahko poljubna, zaradi učinkovitosti pomnilniških dostopov pa se je najbolje držati potenc števila 2.

V datoteki `main.js` sliko najprej prenesemo s strežnika:

```
const imageBitmap = await fetch('image.png')
    .then(response => response.blob())
    .then(blob => createImageBitmap(blob));
```

V zgornji kodi iz strežnikovega odgovora izluščimo vsebino v binarni obliki in jo nato dekodiramo s funkcijo `createImageBitmap`, ki datoteko izbranega slikovnega formata pretvori v nestisnjeno obliko, ki je primerna za prenos na grafično kartico.

Ustvarjanje teksture

Sliko po prenosu s strežnika prenesemo na grafično kartico. V ta namen moramo najprej ustvariti teksturo primerne velikosti in primernega formata, pri tem pa sporočiti še njeno uporabo:

```
const texture = device.createTexture({
    size: [imageBitmap.width, imageBitmap.height],
    format: 'rgba8unorm',
    usage:
        GPUTextureUsage.TEXTURE_BINDING |
        GPUTextureUsage.RENDER_ATTACHMENT |
        GPUTextureUsage.COPY_DST,
});
```

V zgornji kodi velikost teksture preberemo kar iz slike, ki smo jo prenesli s strežnika. Za format smo izbrali `rgba8unorm`, ki teksturi dodeli štiri barvne kanale, pri čemer vsakega od njih predstavimo z 8-bitnim nepredznačenim celim številom. Pripona `norm` bo pomembna v senčilniku, kjer bodo razpon izbranega podatkovnega tipa preslikan v enotski interval v obliki števila s plavajočo vejico. Ker bomo teksturo uporabljali v senčilniku, moramo sporočiti uporabo `TEXTURE_BINDING`. Za prenos slike iz glavnega pomnilnika sta potrebni še zastavici `RENDER_ATTACHMENT` in `COPY_DST`, ki ju potrebuje funkcija `copyExternalImageToTexture`, uporabljena v nadaljevanju.

V naslednjem koraku sliko prenesemo iz glavnega pomnilnika v teksturo v grafičnem pomnilniku:

```
device.queue.copyExternalImageToTexture(
    { source: imageBitmap },
    { texture },
    [imageBitmap.width, imageBitmap.height]);
```

Trije parametri zgornjega ukaza predstavljajo izvor in ponor podatkov ter velikost območja slike, ki ga želimo prenesti. Pri tem imamo natančen nadzor nad območji izvora in ponora ter morebitnimi barvnimi pretvorbami, ki jih po potrebi izvede brskalnik pri prenosu podatkov. V zgornji kodi se v veliki meri zanašamo na privzete nastavitve, ki so prilagojene za prenos 2D barvnih slik.

Vzorčevalnik

Teksturo bomo uporabili v senčilniku fragmentov, kjer bomo iz nje vzorčili barvo. Vzorčenje teksture je prilagodljivo prek vzorčevalnika, ki določa vzorčenje na robovih teksture in interpolacijo podatkov.

Ustvarimo vzorčevalnik s privzetimi nastavitvami:

```
const sampler = device.createSampler();
```

Povezava s senčilnikom

V senčilniku dodamo teksturo in vzorčevalnik kot zunanja vira, tako da jima dodelimo skupino in številko vezave:

```
@group(0) @binding(1) var baseTexture: texture_2d<f32>;
@group(0) @binding(2) var baseSampler: sampler;
```

Uporabili smo kar isto skupino kot za uniformo iz ene od prejšnjih vaj. Tip teksture (`texture_2d<f32>`) odraža njeno razsežnost in podatkovni tip, medtem ko za navadne vzorčevalnike obstaja le en tip (`sampler`).

Teksturo in vzorčevalnik nato vključimo še v skupino vezav in pri tem uporabimo pripadajoče številke vezav:

```
{ binding: 1, resource: texture },
{ binding: 2, resource: sampler },
```

V senčilniku fragmentov namesto interpolirane barve zdaj lahko uporabimo barvo, ki jo vzorčimo iz teksture s funkcijo `textureSample`:

```
output.color = textureSample(baseTexture, baseSampler, vec2(0, 0));
```

Pri tem sporočimo teksturo, ki jo želimo vzorčiti, vzorčevalnik, ki ga želimo pri tem uporabiti, in teksturne koordinate, ki določajo položaj vzorca v teksturnem prostoru. Teksturni prostor je normaliziran, tako da je izhodišče v levem zgornjem kotu teksture, desni spodnji kot pa leži na točki (1, 1).

S temi spremembami bi se morala na zaslonu izrisati enobarvna kocka.

Teksturne koordinate

Teksturne koordinate bo senčilnik fragmentov prejel iz senčilnika oglišč, kamor jih bomo poslali prek atributa. V ta namen bomo nadomestili atribut barve s teksturnimi koordinatami. Najprej spremenimo podatke oglišč:

```
const vertices = new Float32Array([
    // positions         // texcoords
    -1, -1, -1,  1,      0,  0,
    -1, -1,  1,  1,      0,  1,
    -1,  1, -1,  1,      1,  0,
    -1,  1,  1,  1,      1,  1,
     1, -1, -1,  1,      0,  0,
     1, -1,  1,  1,      0,  1,
     1,  1, -1,  1,      1,  0,
     1,  1,  1,  1,      1,  1,
]);
```

V skladu s spremembo podatkov posodobimo razpored podatkov:

```
const vertexBufferLayout = {
    arrayStride: 24,
    attributes: [
        {
            shaderLocation: 0,
            offset: 0,
            format: 'float32x4',
        },
        {
            shaderLocation: 1,
            offset: 16,
            format: 'float32x2',
        },
    ],
};
```

Posodobimo tudi senčilnik, kjer teksturnim koordinatam dodelimo številko atributa 1 in številko interpoliranke 1 ter tip `vec2f`:

```
struct VertexInput {
    @location(0) position: vec4f,
    @location(1) texcoords: vec2f,
}

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(1) texcoords: vec2f,
}

struct FragmentInput {
    @location(1) texcoords: vec2f,
}
```

V senčilniku oglišč spremenimo interpolacijo barv v interpolacijo teksturnih koordinat:

```
output.texcoords = input.texcoords;
```

Interpolirane teksturne koordinate nato lahko uporabimo v senčilniku fragmentov pri vzorčenju teksture:

```
output.color = textureSample(baseTexture, baseSampler, input.texcoords);
```

Z zgornjimi spremembami bi morala biti na zaslonu izrisana teksturirana kocka. Na štirih stranskih ploskvah je del tekstura raztegnjen preko celotne ploskve, kar je posledica slabo nastavljenih teksturnih koordinat. Podatke kocke bi lahko popravili, toda to bi zahtevalo dodajanje novih oglišč in novih indeksov, saj morajo v tem primeru različne ploskve, ki si delijo posamezno oglišče, v tem oglišču uporabljati različne teksturne koordinate. To pa lahko dosežemo le tako, da ustvarimo več oglišč z enakim položajem in različnimi teksturnimi koordinatami. Ker bomo v nadaljevanju podatke o oglišči pridobivali iz zunanjih datotek, se s tem popravkom na tem mestu ne bomo ukvarjali.

Težave z vzorčenjem

Odvisno od ločljivosti teksture in njene vsebine lahko na zaslonu opazimo utripanje barv. Tovrstni artefakti so posledica podvzorčenja, ki ga lahko obvladamo le z odstranitvijo visokih frekvenc iz teksture. Visoke frekvence ustrezajo hitrim spremembam v teksturi, kar lahko opazimo na vsaki meji med teksli.

Težavo lahko do določene mere rešimo z uporabo linearnega filtra:

```
const sampler = device.createSampler({
    minFilter: 'linear',
    magFilter: 'linear',
});
```

Opazimo, da se barve tekslov zdaj prelivajo med seboj, utripanje barv pa ni povsem rešeno. Podvzorčenje je namreč posledica transformacij in predvsem perspektivne projekcije, s katero lahko dosežemo poljubno visoke frekvence pri izrisu na zaslon. Težavo rešimo s piramido slik postopoma nižjih ločljivosti, ki vsebujejo postopoma manj podrobnosti in s tem nižje frekvence.

Piramido slik lahko zgradimo ročno ali samodejno. Ročno pomeni, da posamezen nivo piramide izdelamo z zunanjim programom in ga prenesemo na grafično kartico, samodejno pa pomeni, da nivoje piramide izdelamo kar neposredno na grafični kartici z uporabo senčilnika. Skripta za samodejno generiranje piramide slik je na voljo na tej povezavi. Uporaba je v teh navodilih izpuščena, vsekakor pa je uporaba piramide slik močno priporočljiva.

Naloge

1. Popravi podatke oglišč tako, da bo tekstura pravilno preslikana na vsako ploskev kocke.
2. Napiši funkcijo, ki ustvari teksturo z naključnimi barvami. Ločljivost teksture naj bo podana kot argument funkcije.
3. Program spremeni tako, da bo hkrati uporabljal tako teksture kot barve oglišč. V senčilniku fragmentov naj se barvi zmnožita.
4. Program dopolni tako, da bo pri izrisu uporabljal dve teksturi. V senčilniku fragmentov barvi tekstur povpreči.

Ogrodje

Za lažji razvoj bomo v tej vaji uporabili ogrodje iz repozitorija webgpu-examples. Aplikacijo bomo napisali od začetka, zato datotek s prejšnjih vaj ne potrebujemo. Ogrodje je na voljo v direktoriju `engine`, ki ga skopiramo v svojo aplikacijo, prav tako direktorij `models`. Skopiramo tudi direktorij `lib`, ki vsebuje knjižnice.

Priprava aplikacije

Zdaj ustvarimo glavni datoteki aplikacije. Najprej `index.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Vaja 5</title>
    {
        "imports": {
            "engine/": "./engine/",
            "dat": "./lib/dat.js",
            "glm": "./lib/glm.js"
        }
    }
    </script>
    <link rel="stylesheet" href="engine/style.css">
    <script type="module" src="main.js"></script>
</head>
<body>
    <div class="fullscreen no-touch pixelated">
        <canvas></canvas>
    </div>
</body>
</html>
```

Tako zgrajen HTML bo poskrbel tudi za razteg platna čez celoten zaslon in za pravilno uvažanje modulov.

Nato ustvarimo še `main.js`, kjer postavimo osnovno ogrodje aplikacije in uvozimo vse potrebne razrede:

```
import { ResizeSystem } from 'engine/systems/ResizeSystem.js';
import { UpdateSystem } from 'engine/systems/UpdateSystem.js';
```

```
import {
    Camera,
    Model,
    Node,
    Transform,
} from 'engine/core.js';

const canvas = document.querySelector('canvas');

function update(time, dt) {}
function render() {}
function resize({ displaySize: { width, height }}) {}

new ResizeSystem({ canvas, resize }).start();
new UpdateSystem({ update, render }).start();
```

Branje scene iz datoteke

Prebrali bomo sceno iz datoteke `./models/monkey/monkey.gltf`. Scena je opisana v formatu glTF, ki vsebuje vse informacije o grafu scene, materialih, teksturah in modelih. V aplikaciji jo lahko preberemo z uporabo razreda `GLTFLoader`, ki ga najprej vključimo v aplikacijo:

```
import { GLTFLoader } from 'engine/loaders/GLTFLoader.js';
```

Nato preberemo privzeto sceno in v njej najdemo kamero:

```
const gltfLoader = new GLTFLoader();
await gltfLoader.load('./models/monkey/monkey.gltf');

const scene = gltfLoader.loadScene(gltfLoader.defaultScene);
const camera = scene.find(node => node.getComponentOfType(Camera));
```

V funkciji `resize` poskrbimo za posodabljanje kamere glede na razmerje med širimo in višino zaslona:

```
function resize({ displaySize: { width, height }}) {
    camera.getComponentOfType(Camera).aspect = width / height;
}
```

Upodabljalnik

Sceno lahko izrišemo z uporabo upodabljalnika `UnlitRenderer`. Najprej ga vključimo v aplikacijo:

```
import { UnlitRenderer } from 'engine/renderers/UnlitRenderer.js';
```

Nato ustvarimo upodabljalnik in ga inicializiramo:

```
const renderer = new UnlitRenderer(canvas);
await renderer.initialize();
```

V funkciji `render` pokličemo istoimensko funkcijo upodabljalnika, ki kot parametra sprejema sceno in kamero:

```
function render() {
    renderer.render(scene, camera);
}
```

Na zaslonu se izriše neosvetljen, toda teksturiran 3D model opičje glave.

Interakcija in animacija

Kot zanimivost dodajmo še interakcijo s kamero ter animacijo modela. Za prvo bomo uporabili `OrbitController`, za slednjo pa `RotateAnimator`:

```
import { OrbitController } from 'engine/controllers/OrbitController.js';
import { RotateAnimator } from 'engine/animators/RotateAnimator.js';
```

Razreda instanciramo in objekta pripnemo kot komponenti ustreznim vozliščem:

```
camera.addComponent(new OrbitController(camera, document.body, {
    distance: 8,
}));

const model = scene.find(node => node.getComponentOfType(Model));
model.addComponent(new RotateAnimator(model, {
    startRotation: [0, 0, 0, 1],
    endRotation: [0.7071, 0, 0.7071, 0],
    duration: 5,
```

```
    loop: true,
}));
```

Za pravilno delovanje moramo klicati ustrezne funkcije `update`:

```
function update(time, dt) {
    scene.traverse(node => {
        for (const component of node.components) {
            component.update?.(time, dt);
        }
    });
}
```

Osvetljevanje

Upodabljalnik bomo razširili z Lambertovim osvetlitvenim modelom. Za osnovo bomo uporabili upodabljalnik `UnlitRenderer`, ki ga skopiramo v korenski direktorij aplikacije v datoteko `MyRenderer.js` in nato razred preimenujemo v `MyRenderer`. Potrebujemo tudi senčilnik, ki ga skopiramo iz `UnlitRenderer.wgsl` v korenski direktorij po imenom `MyRenderer.wgsl`. Popravimo poti uvozov in pri tem popravimo še URL, ki ga `MyRenderer` uporablja za dostop do senčilnika.

V datoteki `main.js` uvozimo nov upodabljalnik:

```
import { MyRenderer } from './MyRenderer.js';
```

Nato ga instanciramo namesto `UnlitRenderer`:

```
const renderer = new MyRenderer(canvas);
```

Zdaj lahko upodabljalnik spreminjamo in dopolnjujemo. Najprej bomo dodali osvetljevanje po Lambertovem modelu s konstantno smerjo svetlobe. Lambertov model površino osvetli sorazmerno s kosinusom vpadnega kota svetlobe. Kosinus vpadnega kota lahko enostavno izračunamo s skalarnim produktom, če imamo dostop do normale površine. Normalo dodamo kot atribut oglišč v razporedu, določenem v `MyRenderer.js`:

```
{
    name: 'normal',
    shaderLocation: 2,
    offset: 20,
```

```
    format: 'float32x3',
},
```

Pri tem ne pozabimo spremeniti še velikosti oglišča:

```
arrayStride: 32,
```

Atribut dodamo v senčilnik na lokacijo 2 in poleg tega ustvarimo še interpoliranko na lokaciji 2:

```
struct VertexInput {
    @location(0) position: vec3f,
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}

struct FragmentInput {
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}

struct FragmentOutput {
    @location(0) color: vec4f,
}
```

V senčilniku oglišč normalo transformiramo z normalno matriko (inverz transponirane matrike modela), ki je že na voljo v strukturi `ModelUniforms`:

```
output.normal = model.normalMatrix * input.normal;
```

V senčilniku fragmentov interpolirano normalo pred uporabo normaliziramo, saj je pri linearni interpolaciji prišlo do neizogibne spremembe dolžine vektorja:

```
let N = normalize(input.normal);
```

Določimo še vektor luči, ki naj bo za zdaj konstanten:

```
let L = normalize(vec3f(0, 1, 0));
```

Normalizacija tu sicer ni potrebna, ampak nam omogoča prosto spreminjanje komponent vektorja brez ozira na njegovo dolžino.

S temi informacijami lahko izračunamo Lambertov osvetlitveni faktor:

```
let lambert = max(dot(N, L), 0);
```

S funkcijo `max` se izognemo težavam z negativnim osvetlitvenim faktorjem v primerih, ko vektor luči in normala oklepata topi kot. Z dobljenim osvetlitvenim faktorjem pomnožimo barvne (RGB) komponente izhodne barve:

```
let materialColor = textureSample(baseTexture, baseSampler, input.texcoords) *
let lambertFactor = vec4(vec3(lambert), 1);
output.color = materialColor * lambertFactor;
```

Na zaslonu se izriše osvetljen model.

Točkast svetlobni vir

Usmerjeni svetlobni vir lahko nadomestimo s točkastim, s katerim lažje nadzorujemo videz scene. V senčilniku fragmentov bomo potrebovali položaj fragmenta na površini modela, ki ga lahko zopet pridobimo z interpolacijo, tako da dodamo ustrezno interpoliranko na lokacijo 0 in preimenujemo obstoječo izhodno spremenljivko `position` v `clipPosition` (preimenujemo jo tudi v senčilniku oglišč):

```
struct VertexInput {
    @location(0) position: vec3f,
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}

struct VertexOutput {
    @builtin(position) clipPosition: vec4f,
    @location(0) position: vec3f,
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}

struct FragmentInput {
    @location(0) position: vec3f,
```

```
    @location(1) texcoords: vec2f,
    @location(2) normal: vec3f,
}
```

Položaj v globalnih koordinatah izračunamo z množenjem z matriko modela:

```
output.position = (model.modelMatrix * vec4(input.position, 1)).xyz;
```

Posodobimo izračun vektorja `L` za točkast svetlobni vir:

```
let L = normalize(lightPosition - input.position);
```

Ambientna osvetlitev

Ker je model po dodani osvetlitvi zelo temen, ga lahko dodatno osvetlimo z ambientnim svetlobnim virom. Običajno v računalniški grafiki simuliramo ambientno osvetlitev tako, da osvetlitvenemu faktorju prištejemo ambientni člen:

```
let ambient = 0.3;
let ambientFactor = vec4(vec3(ambient), 1);

output.color = materialColor * (lambertFactor + ambientFactor);
```

S tem je izris precej svetlejši in na videz bolj prijeten.

Na tej točki bi lahko dodali še več parametrov luči, denimo barvo, slabljenje z razdaljo, zrcalne odboje ipd., toda te funkcionalnosti prepustimo kot dodatno vajo.

Komponenta luči

Konstantne parametre v senčilniku bi radi zamenjali z zunanjimi, ki jih lahko enostavneje nadzorujemo, zato bomo ustvarili komponento luči in jo pripeli na novo vozlišče v sceni.

Najprej ustvarimo komponento luči v datoteki `Light.js`:

```
export class Light {

    constructor({
        ambient: 0,
    } = {}) {
        this.ambient = ambient;
```

```
    }

}
```

Položaja luči ni treba zapisovati v komponento, saj je temu namenjena že komponenta `Transform`. Zdaj lahko dodamo luč v sceno:

```
const light = new Node();
light.addComponent(new Transform({
    translation: [3, 3, 3],
}));
light.addComponent(new Light({
    ambient: 0.3,
}));
scene.addChild(light);
```

Da bo zgornja koda delovala, uvozimo razred `Light`.

Luč lahko upodabljalnik sam poišče v sceni ob klicu funkcije `render`. Iz luči moramo nato izluščiti položaj v sceni in prebrati parameter ambientne osvetlitve:

```
const light = scene.find(node => node.getComponentOfType(Light));
const lightComponent = light.getComponentOfType(Light);
const lightMatrix = getGlobalModelMatrix(light);
const lightPosition = mat4.getTranslation(vec3.create(), lightMatrix);
```

Parametre osvetlitve bomo senčilniku podali prek uniforme, ki vsebuje položaj in ambientni faktor:

```
struct LightUniforms {
    position: vec3f,
    ambient: f32,
}
```

Uniformo dodamo kot zunanji vir in ji dodelimo skupino 3 in številko vezave 0:

```
@group(3) @binding(0) var<uniform> light: LightUniforms;
```

Konstantne faktorje lahko zdaj zamenjamo z uniformo:

```
let L = normalize(light.position - input.position);
let ambientFactor = vec4(vec3(light.ambient), 1);
```

V upodabljalniku moramo luči prirediti skupino vezav in medpomnilnik uniform, kar lahko storimo v novi funkciji `prepareLight`:

```
prepareLight(light) {
    if (this.gpuObjects.has(light)) {
        return this.gpuObjects.get(light);
    }

    const lightUniformBuffer = this.device.createBuffer({
        size: 16,
        usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
    });

    const lightBindGroup = this.device.createBindGroup({
        layout: this.pipeline.getBindGroupLayout(3),
        entries: [
            { binding: 0, resource: lightUniformBuffer },
        ],
    });

    const gpuObjects = { lightUniformBuffer, lightBindGroup };
    this.gpuObjects.set(light, gpuObjects);
    return gpuObjects;
}
```

Funkcijo kličemo v funkciji `render`, kjer zapišemo prej pridobljene parametre v medpomnilnik uniform:

```
const { lightUniformBuffer, lightBindGroup } = this.prepareLight(lightComponen
this.device.queue.writeBuffer(lightUniformBuffer, 0, lightPosition);
this.device.queue.writeBuffer(lightUniformBuffer, 12,
    new Float32Array([lightComponent.ambient]));
this.renderPass.setBindGroup(3, lightBindGroup);
```

Animacija luči

Kot zanimivost lahko luč tudi animiramo, saj gre za objekt tipa `Node`, ki vsebuje komponento `Transform`. Dodamo ji lahko denimo linearno gibanje preko komponente `LinearAnimator`. Najprej jo uvozimo:

```
import { LinearAnimator } from 'engine/animators/LinearAnimator.js';
```

Nato komponento dodamo luči:

```
light.addComponent(new LinearAnimator(light, {
    startPosition: [3, 3, 3],
    endPosition: [-3, -3, -3],
    duration: 1,
    loop: true,
}));
```

Vidimo, da se položaj luči spreminja in da se gibanje odraža tudi pri izrisu.

Naloge

1. Program dopolni tako, da bo luč imela tudi barvo.
2. Lambertov osvetlitveni model dopolni s Phongovim modelom za upodabljanje zrcalnih odbojev svetlobe. Poleg položaja luči bo senčilnik potreboval še položaj kamere. Za zrcalni odboj svetlobe si lahko pomagaš s funkcijo `reflect`.
3. Točkasto luč spremeni v reflektorsko, tako da ji dodaš zorni kot. Smer luči je določena z njeno lokalno transformacijo. Senčilnik fragmentov naj glede na zorni kot luči preveri, ali je fragment osvetljen ali ne.
4. Senčilnik posodobi tako, da bo sprejemal 4 luči (uporabi `array`). Upodabljalnik naj v grafu scene poišče 4 luči oz. uniforme manjkajočih luči nastavi tako, da ne vplivajo na osvetlitev.