VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Projekt do předmětu **Základy počítačové grafiky IZG**

Papoušci 2024/2025

2025 Ing. Tomáš Milet, PhD.

Chapter 1

00 Zadání projektu do předmětu IZG.

Vašim úkolem je naimplementovat jednoduchou grafickou kartu (gpu). A dále implementovat funkci pro vykreslení modelů. Všechny soubory, které se vás týkají jsou ve složce studentSolution/src/studentSolution/ V souboru studentSolution/src/studentSolution/gpu.cpp implementujte funkci student_GPU_run - funcionalita vámi implementované grafické karty. V souboru studentSolution/src/studentSolution/prepareModel.cpp implementujete funkce student_prepareModel, student_drawModel_vertexShader a student_drawModel_fragmentShader. Tyto funkce slouží pro zpracování načteného souboru s modelem do paměti grafické karty a command bufferu. Kromě toho se ve složce nachází ještě soubory:

- solutionInterface/src/solutionInterface/gpu.hpp - ten obsahuje deklarace struktur a konstant pro grafickou kartu.
- solutionInterface/src/solutionInterface/modelFwd.hpp - ten obsahuje deklarace struktur a konstant pro model

Chapter 2

01 Jak na projekt?

Projekt se může zdát z prvu obrovský s milioném souborů a všelijakých podivností. Tyto "podivnosti" ale nemusíte řešit. Vše, co se vás týká jsou v podstatě 2 soubory do kterých napíšete váš kód a jeden soubor s deklaracemi struktur pro referenci. Projekt okolo těchto souborů vypadá takto z mnoha důvodů (vytvoření okna, načítaní modelů, testování, ...). A není potřeba se jim zabývat (tedy pokud nechcete vidět vnitřnosti a jak celý projekt funguje). Takže jak na to?
Nejprve si vyzkoušejte, jak by to mělo vypadat...

```
# a mackejte "n" nebo "p" a ovladani mysi
izgProject_windows.exe

# a mackejte "n" nebo "p" a ovladani mysi
./izgProject_linux.bin
```

Jak je to složité? Můj kód pro student_GPU_run má ∼1000 řádků a implementace student_prepareModel a shaderů ∼100 řádku. Není potřeba nic alokovat, pamět je již předchystaná. Takže pokud budete někde volat malloc, new a podobně, zamyslete se. Z C++ se nevyužívá skoro nic (jen vector a knihovna glm, reference). Takže by to mělo jít napsat celkem v pohodě i pro C lidi. Postup řešení:

1. Vyzkoušet si přiložený zkompilovaný referenční projekt izgProject_linux.bin a izgProject_windows.exe. (mačkejte "n" nebo "p", když projekt pustíte, abyste přepínali zobrazované metody).
2. Zprovoznit si překlad
3. Zkusit si projekt pustit a podívat se na parametry příkazové řádky. a jak se aplikace ovládá
4. V projektu jsou přítomy akceptační testy, které vám řeknou, jestli jdete správným směrem a taky vypisují napovědu.
5. Začít implementovat funkci student_GPU_run a kontrolovat váš postup podle přiložených testů.
6. Začít implementovat funkci student_prepareModel
7. Začít implementovat funkci student_drawModel_vertexShader
8. Začít implementovat funkci student_drawModel_fragmentShader
9. Ověřte si implementaci na Merlinovi
10. Odevzdejte
11. ???
12. profit

Každý úkol má přiřazen akceptační test, takže si můžete snadno ověřit funkčnosti vaší implementace.

Úkoly lze rozdělit do dvou částí: implementace grafické karty a implementace kreslení modelů se stíny.

Chapter 3

02 Grafická karta, paměť a příkazy

3.1 Teorie

První věc, na co se asi ptáte: "Jak vypadá počítač", "Jak vypadá grafická karta, jak se s ní komunikuje a co je její chování?"
Cílem této části je tvorba grafické karty. Chtěli jste si někdy vytvořit grafickou kartu? Ne? A chcete alespoň vědět, jak se vykreslují počítačové hry a jak funguje svět real-time počítačové grafiky? Nebo alespoň chcete vědět, jak nevyletět u státnic? Základem je počítač s procesorem a grafickou kartou: Jak je vidět, tak s grafickou kartou se komunikuje pomocí fronty příkazů (v tomto projektu fronta není), po které se posílají balíčky práce. Balíček práce (CommandBuffer) v sobě obsahuje mnoho úkolů, které má grafická karta provést. Koncept command bufferu lze najít například ve Vulkánu: `CommandBuffer`. Balíček práce se vždy provede nad pamětí grafické karty. Toto fungování grafické karty je zajištěno (bude, až to naprogramujete) funkcí student_GPU_run.

Funkce student_GPU_run se nachází v souboru studentSolution/src/studentSolution/gpu.cpp. Je to funkce, která reprezentuje chování vaší grafické karty. Lze pomocí ní kreslit trojúhelníky, mazat framebuffer, nastavovat číslo vykreslovacího příkazu nebo nastavovat aktivní objekty a další.

```
void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  (void)mem;
  (void)cb;
}
```

Vašim úkolem je ji postupně naprogramovat. Na jeden pokus ji nenaprogramujete, budete ji programovat postupně. Doporučuji si kousky funkce dávat do vlastních podfunkcí, ať máte kód přehledný.

Funkce student_GPU_run bere dva vstupní parametry:

- paměť grafické karty GPUMemory, nad kterou jsou vykonávány všechny operace,
- CommandBuffer - seznam operací k provedení.

Paměť grafické karty: Výpis GPUMemory ze souboru solutionInterface/src/solutionInterface/gpu.hpp

```
struct GPUMemory{
  uint32_t         maxUniforms          = 0      ;
  uint32_t         maxVertexArrays      = 0      ;
  uint32_t         maxTextures          = 0      ;
  uint32_t         maxBuffers           = 0      ;
  uint32_t         maxPrograms          = 0      ;
  uint32_t         maxFramebuffers      = 0      ;
  uint32_t         defaultFramebuffer   = 0      ;
  Buffer          *buffers              = nullptr;
  Texture         *textures             = nullptr;
  Uniform         *uniforms             = nullptr;
  Program         *programs             = nullptr;
  Framebuffer     *framebuffers         = nullptr;
  VertexArray     *vertexArrays         = nullptr;
  uint32_t         activatedFramebuffer = 0      ;
```

```
  uint32_t         activatedProgram     = 0      ;
  uint32_t         activatedVertexArray = 0      ;
  uint32_t         gl_DrawID            = 0      ;
  StencilSettings  stencilSettings                ;
  BlockWrites      blockWrites                    ;
  BackfaceCulling  backfaceCulling                ;

  //Do not worry about these.
  //This is just to suppress valgrind warnings because of the large stack.
  //Otherwise everything would be placed on the stack and not on the heap.
  //I had to allocated this structure on the heap, because it is too large.
  GPUMemory();
  GPUMemory(GPUMemory const&o);
  ~GPUMemory();
  GPUMemory&operator=(GPUMemory const&o);
};
```

Operace v command bufferu: Výpis CommandBuffer ze souboru solutionInterface/src/solutionInterface/gpu.hpp

```
struct CommandBuffer{
  uint32_t static const maxCommands           = 10000;
  uint32_t              nofCommands           = 0    ;
  Command               commands[maxCommands]        ;
};
```

Jak můžete vidět, obsahuje tři položky: maximální počet příkazů, který může být uložen, počet uložených příkazů a samotné příkazy.
Vaše grafická karta by měla umožnit několik druhů práce:

- navázání aktivního framebufferu,
- navázání aktivního shader programu,
- navázání aktivního vertex array objektu,
- nastavení, jestli je zakázáno zapisovat do framebufferu,
- nastavení, jestli se mají odvrácené trojúhelníky zahazovat,
- nastavení, která strana trojúhelníku je přivrácená ke kameře,
- nastavení stencilových operací a testu,
- nastavení čísla kreslícího příkazu,
- vyvolání uživatelské funkce,
- čistění paměti barvy ve framebufferu,
- čistění paměti hloubky ve framebufferu,
- čistění paměti stencilu ve framebufferu,
- kreslení do framebufferu,
- sub command.

Struktura samotného příkazu vypadá takto:

```
struct Command{
  CommandData data                    ;
  CommandType type = CommandType::EMPTY;
};
```

Je složena z typu a dat. Typ je enum:

```
enum class CommandType{
  EMPTY                          ,
  BIND_FRAMEBUFFER               ,
  BIND_PROGRAM                   ,
  BIND_VERTEXARRAY               ,
  BLOCK_WRITES_COMMAND           ,
  SET_BACKFACE_CULLING_COMMAND,
  SET_FRONT_FACE_COMMAND         ,
  SET_STENCIL_COMMAND            ,
  SET_DRAW_ID                    ,
  USER_COMMAND                   ,
```

```
  CLEAR_COLOR                ,
  CLEAR_DEPTH                ,
  CLEAR_STENCIL              ,
  DRAW                       ,
  SUB_COMMAND                ,
};
```

A data je union:

```
union CommandData{
  CommandData():drawCommand(){}
  BindFramebufferCommand    bindFramebufferCommand   ;
  BindProgramCommand        bindProgramCommand       ;
  BindVertexArrayCommand    bindVertexArrayCommand   ;
  BlockWritesCommand        blockWritesCommand       ;
  SetBackfaceCullingCommand setBackfaceCullingCommand;
  SetFrontFaceCommand       setFrontFaceCommand      ;
  SetStencilCommand         setStencilCommand        ;
  SetDrawIdCommand          setDrawIdCommand         ;
  UserCommand               userCommand              ;
  ClearColorCommand         clearColorCommand        ;
  ClearDepthCommand         clearDepthCommand        ;
  ClearStencilCommand       clearStencilCommand      ;
  DrawCommand               drawCommand              ;
  SubCommand                subCommand               ;
};
```

Union je něco jako struktura až na to, že jeho velikost je daná největší komponentou. Data unionu jsou uložena přes sebe a je možné uložit jen jednu komponentu. Vzhledem k tomu, že jsem projekt psal v C++, je přítomen i konstruktor, ale toho si nemusíte všímat, jen udává, na co bude union inicializovaný - na draw command.

Chapter 4

03 Úkoly ohledně Command Bufferu

V tomto projektu musíte naimplementovat vlastní grafickou kartu. Tu budete implementovat v souboru↩ : studentSolution/src/studentSolution/gpu.cpp

4.1 Teorie: Nastavování aktivních objektů GPU a zápisu nastavení

Grafická karta obsahuje mnoho objektů. Jsou to:

- textury,
- buffery,
- programy,
- framebuffery,
- vertex array objekty,
- uniformy a další.

Některé z těchto objektů se přímo využívají při kreslení. Jsou to objekty:

- framebuffery,
- programy,
- vertex array objekty.

Je potřeba vědět, kam se kreslí (Framebuffer), je potřeba vědět jak se kreslí (Program a nastavení v objektech: BlockWrites, StencilSettings a BackfaceCulling) a je potřeba vědět, odkud se berou data pro kreslení (VertexArray). Síla dnešních grafických karet spočívá v jejich programovatelnosti a široké nastavitelnosti. Dnes je možné kreslit to vícero framebufferů, využívat k tomu tisíce programů a mít k tomu milióny objektů. Proto je nutné grafické kartě říct, které objekty jsou v danou chvíli aktivní. V OpenGL se to provádí příkazy:

- `glBindFramebuffer(...)`
- `glUseProgram(...)`
- `glBindVertexArray(...)`

Těmto příkazům odpovídají v tomto projetku příkazy BindFramebufferCommand, BindProgramCommand, BindVertexArrayCommand.

Stejně tak je potřeba GPU říct, jaké je nastavení pro kreslení. Například se může zakázat zápis barvy, hloubky a stencilové hodnoty do framebuffer, nebo se může zapnout ořezávání odvrácených stran trojúhelníků, nebo se mohou nastavit stencilové operace. V OpenGL by to byly tyto příkazy:

- Zakázání / Povolení zápisu do framebufferu (BlockWritesCommand):
  - `glColorMask(...)`
  - `glDepthMask(...)`
  - `glStencilMask(...)`
- Povolení / zakázání zahazování odvrácených stran trojúhelníků (SetBackfaceCullingCommand):
  - `glEnabled(GL_CULL_FACE)`
  - `glDisable(GL_CULL_FACE)`
- Určení, co je to přivrácená strana trojúhelníku (SetFrontFaceCommand):
  - `glFrontFace(GL_CW)`
  - `glFrontFace(GL_CCW)`
- Stencilové nastavení (SetStencilCommand):
  - `glStencilFunc(...)`
  - `glStencilOpSeparate(...)`
  - `glEnabled(GL_STENCIL_TEST)`
  - `glDisabled(GL_STENCIL_TEST)`

Podobně je to ve Vulkánu, ale je to tam složitější.
Těmto nastavením odpovídají v tomto projektu příkazy BlockWritesCommand, SetBackfaceCullingCommand, SetFrontFaceCommand a SetStencilCommand.

4.2 Úkol 0: Aktivování objektů a nastavení

Vašim prvním úkolem bude správně vybírat aktivní objekty na grafické kartě a zápis nastavení GPU. Vážou se k tomu tyto testy:

```
./izgProject -c --test 7 --up-to-test
```

Editujte funkci student_GPU_run v souboru studentSolution/src/studentSolution/gpu.cpp.

4.2.1 Test 0 - bindFramebuffer

```
./izgProject -c --test 0
```

Tento test zkouší, zda funguje command BindFramebufferCommand

```
struct BindFramebufferCommand{
  uint32_t id = 0;
};
```

Pokud se v command bufferu objeví tento příkaz, je nutné nastavit aktivní framebuffer v paměti gpu↩ : GPUMemory::activatedFramebuffer. Pamatujte, je potřeba zpracovat CommandBuffer a správně reagovat na příkazy, které jsou v něm uloženy.

```
void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  // průchod všemi příkazy v CommandBufferu
  for(uint32_t i=0;i<cb.nofCommands;++i){
    // typ a data příkazu
    CommandType type = cb.commands[i].type;
    CommandData data = cb.commands[i].data;
    if(type == CommandType::BIND_FRAMEBUFFER){ // pokud je to BindFramebufferCommand
      //nastav aktivní framebuffer
      mem.activatedFramebuffer = data.bindFramebufferCommand.id;
    }
  }
}
```

4.2.2 Test 1 - bindProgram

```
./izgProject -c --test 1
```

Tento test zkouší, zda funguje command BindProgramCommand

```
struct BindProgramCommand{
  uint32_t id = 0;
};
```

Pokud se v command bufferu objeví tento příkaz, je nutné nastavit aktivní program v paměti gpu↩ : GPUMemory::activatedProgram.

4.2.3 Test 2 - bindVertexArray

```
./izgProject -c --test 2
```

Tento test zkouší, zda funguje command BindVertexArrayCommand

```
struct BindVertexArrayCommand{
  uint32_t id = 0;
};
```

Pokud se v command bufferu objeví tento příkaz, je nutné nastavit aktivní vertex array v paměti gpu↩ : GPUMemory::activatedVertexArray.

4.2.4 Test 3 - blockWrites

```
./izgProject -c --test 3
```

Tento test zkouší, zda funguje command BlockWritesCommand

```
struct BlockWritesCommand{
  BlockWrites blockWrites;
};
struct BlockWrites{
  bool color   = false;
  bool depth   = false;
  bool stencil = false;
};
```

Pokud se v CommandBuffer(u) objeví tento příkaz, je nutné nastavit v paměti gpu: GPUMemory::blockWrites.

4.2.5 Test 4 - setBackface

```
./izgProject -c --test 4
```

Tento test zkouší, zda funguje command SetBackfaceCullingCommand

```
struct SetBackfaceCullingCommand{
  bool enabled = false;
};
```

Pokud se v CommandBuffer(u) objeví tento příkaz, je nutné nastavit v paměti gpu : GPUMemory::backfaceCulling položku BackfaceCulling::enabled.

```
struct BackfaceCulling{
  bool enabled                    = false;
  bool frontFaceIsCounterClockWise = true ;
};
```

4.2.6 Test 5 - setFrontFace

```
./izgProject -c --test 5
```

Tento test zkouší, zda funguje command SetFrontFaceCommand

```
struct SetFrontFaceCommand{
  bool frontFaceIsCounterClockWise = true;
};
```

Pokud se v CommandBuffer(u) objeví tento příkaz, je nutné nastavit v paměti gpu : GPUMemory::backfaceCulling položku BackfaceCulling::frontFaceIsCounterClockWise

```
struct BackfaceCulling{
  bool enabled                    = false;
  bool frontFaceIsCounterClockWise = true ;
};
```

4.2.7 Test 6 - setStencil

```
./izgProject -c --test 6
```

Tento test zkouší, zda funguje command SetStencilCommand

```
struct SetStencilCommand{
  StencilSettings settings;
};
struct StencilSettings{
  bool        enabled  = false             ;
  StencilFunc func     = StencilFunc::ALWAYS;
  uint32_t    refValue = 0                 ;
  StencilOps  frontOps                     ;
  StencilOps  backOps                      ;
};
```

Pokud se v CommandBuffer(u) objeví tento příkaz, je nutné nastavit v paměti gpu : GPUMemory::stencilSettings.

```
struct StencilSettings{
  bool        enabled  = false             ;
  StencilFunc func     = StencilFunc::ALWAYS;
  uint32_t    refValue = 0                 ;
  StencilOps  frontOps                     ;
  StencilOps  backOps                      ;
};
struct StencilOps{
  StencilOp sfail  = StencilOp::KEEP;
  StencilOp dpfail = StencilOp::KEEP;
  StencilOp dppass = StencilOp::KEEP;
};
enum class StencilOp{
   KEEP     ,
   ZERO     ,
   REPLACE  ,
   INCR     ,
   INCR_WRAP,
   DECR     ,
   DECR_WRAP,
   INVERT   ,
};
enum class StencilFunc{
  NEVER   ,
  LESS    ,
  LEQUAL  ,
  GREATER ,
  GEQUAL  ,
  EQUAL   ,
  NOTEQUAL,
  ALWAYS  ,
};
```

4.2.8 Test 7 - setDrawID

```
./izgProject -c --test 7
```

Tento test zkouší, zda funguje command SetDrawIdCommand

```
struct SetDrawIdCommand{
  uint32_t id = 0;
};
```

Pokud se v CommandBuffer(u) objeví tento příkaz, je nutné nastavit v paměti gpu : GPUMemory::gl_DrawID.

Chapter 5

04 Čistící příkazy

5.1 Teorie: Čistění framebufferu

Framebuffer je složen ze tří bufferů: paměť barvy (color buffer), paměť hloubky (depth buffer) a paměť stencilu (stencil buffer): Všechny mají stejné rozlišení. Barevný buffer má několik kanálů (až čtyři), každý má stejnou velikost a typ. Hlouboký buffer má hloubku uloženou ve floatech. Stencilový buffer má hodnotu uloženou v 8bitovém čísle. Framebuffer je koncipován tak, že pixel na souřadnicích [0,0] je v levém dolním rohu, osa X je doprava a oxy Y nahoru. Je možné jej přetočit vzhůru nohama pomocí příznaku Framebuffer::yReversed.

Všechny framebuffery se nachází v paměti grafické karty (GPUMemory):

```
struct Framebuffer{
  uint32_t width     = 0    ;
  uint32_t height    = 0    ;
  bool     yReversed = false;
  Image    color             ;
  Image    depth             ;
  Image    stencil           ;
};
```

Framebuffer je poměrně složitá struktura. Je složena ze:

- tři Image - barva, hloubka, stencil,
- šířka,
- výška,
- yReversed - v případě, že je framebuffer vzhůru nohama.

Image je struktura obsahující 2D data. Je využívána u framebufferů a textur.

```
struct Image{
  enum Channel{
    RED   = 0,
    GREEN    ,
    BLUE     ,
    ALPHA    ,
  };
  enum Format{
    U8 ,
    F32,
  };
  void*    data            = nullptr               ;
  uint32_t channels        = 4                     ;
  Format   format          = U8                    ;
  uint32_t pitch           = 0                     ;
  uint32_t bytesPerPixel   = 0                     ;
  Channel  channelTypes[4] = {RED,GREEN,BLUE,ALPHA};
};
```

Image je inspirovaný strukturami `SDL_Surface` a `SDL_PixelFormat`.
Struktura obsahuje několik položek:

- Image::data - ukazatel na začátek,
- Image::channels - počet kanálů,
- Image::format - formát kanálů,
- Image::pitch - šířka řádku v bajtech,
- Image::bytesPerPixel - velikost jednoho pixelu v bajtech,
- Image::channelTypes - tabulka mapování čísla kanálu na typ kanálu.

Adresování dat může být poněkud komplikované...
```
// Pixel [x,y] začíná na adrese:
uint8_t* pixelStart = ((uint8_t*)data) + y*pitch + x*bytesPerPixel;

// Pokud jsou data typu float
if(format == Image::FLOAT32){
  float*pixelf = (float*)pixelStart;

  // Kanál 0 odpovídá barvě channelTypes[0]
  // tzn. 0 nemusí být RED
  pixelf[0] = 0.5f;
}
// Pokud jsou data typu uint8_t
if(format == Image::UINT8){
  uint8_t*pixelu = (uint8_t*)pixelStart;
  pixelu[0] = 127;
}
```

Čistící příkazy (ClearColorCommand, ClearDepthCommand, ClearStencilCommand) vypadají takto:
```
struct ClearColorCommand{
  glm::vec4 value = glm::vec4(0);
};
struct ClearDepthCommand{
  float value = 1e10;
};
struct ClearStencilCommand{
  uint8_t value = 0u;
};
```

Čistící příkazy obsahují hodnotu, na kterou se mají vyčistit barevný, hloubkový nebo stencilový buffer. Všimněte si, že barva je uložena jako floatový vektor glm::vec4. V tomto vektoru je barva v rozsahu [0,1] typu float. Čistící barvu musíte z toho rozsahu převést na správný typ podle typu barevného bufferu.

5.2 Úkol 1: Čištění framebufferu

Vašim úkolem bude naprogramovat obsluhu čistících příkazů. K tomuto úkolu se vážou testy:

```
./izgProject -c --test 10 --up-to-test
```

Takto vypadá pseudokód, jak můžete začít psát:
```
void clearColor(GPUMemory&mem,ClearColorCommand cmd){
  // ukázka čistícího příkazu

  // výběr framebufferu
  Framebuffer*fbo = mem.framebuffers+mem.activatedFramebuffer;

  // obsahuje framebuffer barevný buffer?
  if(fbo->color.data){
    for(uint32_t y=0;y<fbo->height;++y)
      for(uint32_t x=0;x<fbo->width;++x){
        void*pixelStart = getPixel(fbo->color,x,y);
        for(uint32_t i=0;i<fbo->color.channels;++i){
          //...
        }
      }
  //...
  }
```

```
}

void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  for(uint32_t i=0;i<cb.nofCommands;++i){
    CommandType type = cb.commands[i].type;
    CommandData data = cb.commands[i].data;
    if(type == CommandType::CLEAR_COLOR)
      clearColor(mem,data.clearColorCommand);
  }
}
```

5.2.1 Test 8 - čištění framebufferu

```
./izgProject -c --test 8
```

Tento test zkouší vyčistit framebuffer.

5.2.2 Test 9 - čištění částečného framebufferu

```
./izgProject -c --test 9
```

Tento test zkouší vyčistit částečně specifikovaný framebuffer. Paměť barvy, paměť hloubky i stencil může být prázdná (nullptr), v takovém případě čištění neproběhne.

5.2.3 Test 10 - zápis do vícero framebufferů

```
./izgProject -c --test 10
```

Tento test zkouší čistit různé Framebuffer(y), ne jen nultý. Čistící příkaz čistí aktivní framebuffer.

Chapter 6

05 Uživatelský příkaz, číslování a podcommand buffer

6.1 Úkol 2: - Uživatelský příkaz a číslování

Cílem této části je zprovoznit uživatelský příkaz, kreslící příkazy a pod CommandBuffer(y).

6.1.1 Test 11 - UserCommand

```
./izgProject -c --test 11
```

Tento test zkouší, zda jste naimplementovali obsluhu uživatelského příkazu: UserCommand.

```
struct UserCommand{
  UserCommandFce callback = nullptr;
  void*          data     = nullptr;
};
```

Pokud grafická karta narazí na tento příkaz, měla by vyvolat callback a dát mu data. Pozor, uživatelský callback může být nullptr, v takovém případě se příkaz ignoruje.

6.1.2 Test 12 - DrawCommand a počítání gl_DrawID

```
./izgProject -c --test 12
```

Tento test zkouší, zda jste naimplementovali obsluhu kreslícího příkazu: DrawCommand.

```
struct DrawCommand{
  uint32_t nofVertices     = 0    ;
};
```

Pokud grafická karta narazí na tento příkaz, měla by spustit kreslení. Každý vykreslovací příkaz je číslovaný vzestupně od počátku spuštění. Cílem tohoto testu je ověřit, že počítáte vykresovací příkazy (a nastavujete GPUMemory::gl_DrawID). Toto číslování se používá pro výběr materiálů, textur, modelových matic a podobně. Číslují se jen vykreslovací příkazy. Pokud je mezi kreslícími jiný příkaz, neovlivní to číslování. Výjimkou je příkaz SetDrawIdCommand, který umožňuje explicitně GPUMemory::gl_DrawID nastavit. Hrubý pseudokód může vypadat nějak takto:

```
void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  // smyčka přes příkazy
  for(... commands ...){
    // vykreslovací příkaz
    if (commandType == CommandType::DRAW ){
      // kresli
      draw(mem, drawCommand);
      // počítadlo kreslících příkazů
      mem.gl_DrawID++;
    }
    if (commandType == CommandType::SET_DRAW_ID){
      mem.gl_DrawID = ...
    }
  }
}
```

6.1.3 Test 13 - sub command

Testy:
```
./izgProject -c --test 13
```

SubCommand je způsob, jak rozšiřovat a větvit CommandBuffer. SubCommand obsahuje ukazatel na další CommandBuffer.
```
struct SubCommand{
  CommandBuffer*commandBuffer = nullptr;
};
```

Koncept je podobný jako u sekundárních command bufferů ve Vulkánu: Sekundární Command Buffer. Příkladem využití SubCommand v tomto projektu je vykreslování stínů. Technika vykreslování stínů vyžaduje vykreslit scénu 2x, pokaždé s jiným programem a framebufferem. Je tak možné uložit si sub CommandBuffer pro vykreslení scény a ten pak dvakrát uložit v primárním CommandBuffer(u) pro celý snímek. SubCommand může být vložen v libovolné hloubce, tzn. sub command buffer může obsahovat sub command buffer. Hrubý pseudokód může vypadat nějak takto:
```
void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  // smyčka přes příkazy
  for(... commands ...){
    // sub command
    if (commandType == CommandType::SUB_COMMAND){
      processSubCommandRecursive(commandData.subCommand.commandBuffer);
    }
  }
}
```

Chapter 7

06 Teorie o vektorové a rastrové části GPU

7.1 Úvod

Grafická karta je navržena tak, aby se minimalizovaly přenosy CPU <-> GPU. Je to z toho důvodu, že PCIe sběrnice je oproti všem zůčastněným částem při kreslení nejpomalejší. Snažíme se o to, aby se používalo menší množství větších přenosů data mezi CPU <-> GPU. Velké množství malých přenosů je neefektivní a způsobuje čekání jak na straně CPU, tak na straně GPU. Další věcí, která způsobuje zpomalování kreslení je velké množství samostatných vykresovacích příkazů. Je lepší jedním příkazem vykreslit milión trojúhelníků než miliónem příkazů vykreslit stejný milión trojúhelníků po jednom. Z těchto důvodů vzniky command buffery a další techniky. Dnes je možné pomocí jednoho příkazu vykreslit celou scénu i s mnoha efekty. Příkladem nechť je funkce z OpenGL `glDrawElementsIndirect`. Nastává však jeden problém. Pokud se vše vykreslí pomocí jednoho volání, jak se každému objektu nastaví správná barva, pozice a materiál?
V OpenGL i ve Vulkánu se to řeší pomocí číslování vykreslovacích příkazů `gl_DrawID`. Pomocí tohoto čísla je možné typicky v shader programu vybrat správnou modelovou matici, materiál a jiné vlastnosti. Ale co je to vůbec shader program, kde se berou data a jak vůbec funguje vykresování? A co je to vykreslovací řetězec a jak funguje?

Cílem následujícího výkladu je přiblížit fungování grafické karty.

7.2 Grafická karta

Hlavním účelem grafické karty je převod vektorové grafiky na rastrovou. Data se čtou z paměti, pak se zpracují zobrazovacím řetězcem (ve kterém běží programy) a výsledek se opět zapíše do paměti. Zobravací řetězec je složitý, lze rozdělit na tři části: vektorová část, rasterizace a rastrová část. Akce/příkaz kreslení operuje nad pamětí: Příkaz kreslení je prováděn stejně jako příkaz čistění v grafické kartě. Proces kreslení na grafické kartě probíhá v zobrazovacím řetězci.

7.3 Zobrazovací řetězec

Zobrazovací řetězec je složen ze tří částí: vektorová část, rasterizace, rastrová část. Úkolem vektorové části je transformovat vektorovou grafiku, posouvat trojúhelníky a podobně. Úkolem raterizace je vektorové elementy převést na rastr. Úkolem frastrové části je obarvit vyrastrované vektory.
Část rasterizace a dál nás v tomto úkolu nezajímá, to až později. Tento test je zaměřený na vektorovou část a to jen na její vstup a vertex shader.

7.4 Vektorová část zobrazovacího řetězce

Vektorová část zobrazovacího řetězce se dá rozdělit přibližně na dvě části:

- Část Vertexů
- Část Primitiv

Následující výklad a testy projektu se budou týkat právě vektorové části zobrazovacího řetězce.

Chapter 8

07 Vektorová čast GPU: část vertexů

8.1 Vertexová část zobrazovacího řetězce

Cílem vektorové části je zpravovávat vektorovou grafiku: body, trojúhelníky. Většinou se tím myslí: čtení z paměti a sestavení vrcholů, vyvolání vertex shaderu nad každým vrcholem, sestavení trojúhelníků, ořez, perspektivní dělení a připravení pro rasterizaci (viewport transformace). Rasterizace rasterizuje připravené trojúhelníky a produkuje fragmenty (čtvercové úlomky trojúhelníku, které se nakonec zapíšou do framebufferu). Cílem rastrové části je obarvit tyto fragmenty pomocí fragment shaderu, odfiltrovat fragmenty, které jsou příliš daleko (depth test) a smíchat je s framebufferem (blending).
Ze začátku implementace kreslení se budete zabývat pouze vektorovou částí - a to částí před vertex shaderem (včetně). Vertex assembly jednota se stará o sestavování vrcholů. Vertex processor tyto vrcholy "prožene" uživatelem specifikovaným vertex shaderem. Část za vertex shaderem se stará o sestavení trojúhelníku, jeho ořezu a ztransformování pro rasterizaci.

8.1.1 Neindexované a indexované kreslení

Existují dva druhy vykreslování:

- neindexované,
- indexované.

Indexované kreslení je způsob snížení redundance dat s využitím indexů na vrcholy.

Vrcholy jsou během kreslení číslovány pomocí čísla InVertex::gl_VertexID. InVertex::gl_VertexID je unikátní číslo vrcholu do paměti vertexů, na jehož základě pracuje Vertex Assembly jednotka.

8.1.2 Vertex Assembly jednotka

Vertex Assembly (nebo taky Vertex Puller, Vertex Specification, ...) je zařízení na grafické kartě, které se stará o sestavení vrcholů.

Vertex není jen bod v prostoru. Vertex je uživatelem specifikovaná struktura. Uživatel může chtít do vrcholů uložit různá data, proto do nich může přidat vertex attributy. Kromě uživatelem specifikovaných atributů, obsahují i pevně vestavěné atributy (gl_VertexID a další).

Sestavené vcholy jsou posílány do vertex shaderu pro zpracování uživetelem definovaným kódem. Vertex shader transformuje vrcholy maticemi a vypočítává výstupní vrcholy. Vrchol (InVertex) je složen z maxAttribs vertex atributů, každý může být různého typu (AttribType (float, vec2, vec3, vec4, ...) a čísla vrcholu InVertex::gl_VertexID.

8.1.3 Tabulka nastavení Vertex Array

Vertex Assembly jednotka se řídí podle nastavení ze struktury VertexArray.

Vertex Assembly jednotka je složena z maxAttribs čtecích hlav, které sestavují jednotlivé vertex attributy. InVertex je složen z maxAttribs atributů, každý odpovídá jedné čtecí hlavě z Vertex Assembly jednotky. Čtecí hlava obsahuje nastavení - offset, stride, type a buffer. Pokud je čtecí hlava povolena (typ není empty), měla by zkopírovat data (o velikosti vertex atributu) z bufferu od daného offsetu, s krokem stride. Všechny velikosti jsou v bajtech. Krok se použije při čtení různých vrcholů: atributy by měly být čteny z adresy: buf_ptr + offset + stride∗gl_VertexID
Na dalších dvou obrázcích je příklad stavu Vertex Assembly jednotky ve dvou (0. a 1.) invokaci vertex shaderu.

8.1.4 Vertex Processor

Úkolem vertex processoru je pouštět uživatelem specifikovaný vertex shader. Obvykle provádí transformace vrcholů pomocí transformačních matic. Vertex processor vykonává shader (kus programu), kterému se říká vertex shader. Vstupem vertex shaderu je vrchol InVertex, výstupem je vrchol OutVertex. Dalším (konstatním) vstupem vertex shaderu jsou uniformní proměnné a textury ShaderInterface, které jsou uložené v rámci shader programu. Pokud se uživatel rozhodne vykreslit 5 trojúhelníků je vertex shader spuštěn $5 \cdot 3 = 15$. Jednotlivé spuštění (invokace) vertex shaderu vyžadují nové vstupní vrcholy a produkují nové výstupní vrcholy. To ve výsledku znamená, že se pro každou invokaci vertex shaderu spustí Vertex Assembly jednotka, která sestaví vstupní vrchol.

8.2 Úkol 3: kreslící příkaz - vertexová část GPU

Cílem této sekce je obsloužit vertexovou část GPU. Do této části spadá: Vertex Array, Vertex Shader, Buffery, Indexy, gl_VertexID, gl_DrawID a čtení z bufferů. Vážou se k tomu tyto testy:

```
./izgProject -c --test 21 --up-to-test
```

Opět editujete funkci student_GPU_run v souboru studentSolution/src/studentSolution/gpu.cpp.

8.2.1 Test 14 - spouštění vertex shaderu

Úkol je zprovoznit spouštění vertex shaderu. K tomtu se váže test:

```
./izgProject -c --test 14
```

Při kreslení musíte zavolat vertex shader tolikrát, kolik je zadáno v kreslícím příkazu (DrawCommand). Kreslící příkaz je struktura:

```
struct DrawCommand{
  uint32_t nofVertices     = 0    ;
};
```

Struktura obsahuje počet vertexů pro vykreslení. Program, který by se pro kreslení měl využít se nachází v paměti grafické karty GPUMemory.

```
struct GPUMemory{
  uint32_t         maxUniforms          = 0      ;
  uint32_t         maxVertexArrays      = 0      ;
  uint32_t         maxTextures          = 0      ;
  uint32_t         maxBuffers           = 0      ;
  uint32_t         maxPrograms          = 0      ;
  uint32_t         maxFramebuffers      = 0      ;
  uint32_t         defaultFramebuffer   = 0      ;
  Buffer          *buffers              = nullptr;
  Texture         *textures             = nullptr;
  Uniform         *uniforms             = nullptr;
  Program         *programs             = nullptr;
  Framebuffer     *framebuffers         = nullptr;
  VertexArray     *vertexArrays         = nullptr;
```

```
  uint32_t         activatedFramebuffer = 0      ;
  uint32_t         activatedProgram     = 0      ;
  uint32_t         activatedVertexArray = 0      ;
  uint32_t         gl_DrawID            = 0      ;
  StencilSettings  stencilSettings                ;
  BlockWrites      blockWrites                    ;
  BackfaceCulling  backfaceCulling                ;

  //Do not worry about these.
  //This is just to suppress valgrind warnings because of the large stack.
  //Otherwise everything would be placed on the stack and not on the heap.
  //I had to allocated this structure on the heap, because it is too large.
  GPUMemory();
  GPUMemory(GPUMemory const&o);
  ~GPUMemory();
  GPUMemory&operator=(GPUMemory const&o);
};
```

Správný program je vybrán pomocí čísla aktivního programu GPUMemory::activatedProgram. Program je opět struktura:

```
struct Program{
  VertexShader   vertexShader   = nullptr;
  FragmentShader fragmentShader = nullptr;
  AttribType  vs2fs[maxAttribs] = {AttribType::EMPTY};
};
```

Struktura programu obsahuje vertex shader. Vertex shader je v ukazatel na funkci. Na normálním GPU se jedná o program (třeba v GLSL), který se kompiluje. V tomto projektu je to C/C++ funkce, která je uložena v ukazateli na funkci. Vertex shader bere 3 parametry

```
using VertexShader = void(*)(
    OutVertex            &outVertex,
    InVertex        const&inVertex ,
    ShaderInterface const&si       );
```

V tomto testu byste měli správně nastavit InVertex::gl_VertexID. Obdobně jako číslování kreslících příkazů, existuje i číslování vrcholů. Zatím bude stačit pořadové číslo vrcholu. Vstupní vrchol se nachází ve struktuře InVertex

```
struct InVertex{
  Attrib   attributes[maxAttribs]    ;
  uint32_t gl_VertexID           = 0;
};
```

8.2.2 Test 15 - test číslování vykreslovacích příkazů

Úkol je zprovoznit proměnnou ShaderInterface::gl_DrawID. Vertex shader obdrží mimo vstupního vertexu ještě ShaderInterface. ShaderInterface obsahuje konstanty.

```
struct ShaderInterface{
  Uniform const*uniforms  = nullptr;
  Texture const*textures  = nullptr;
  uint32_t      gl_DrawID = 0      ;
};
```

K tomtu se váže test:

```
./izgProject -c --test 15
```

Vertex shader by měl vědět, v rámci jakého vykreslovacího příkazu byl puštěn.

8.2.3 Test 16 - test proloženého kreslení a čistění

Tento test zkouší do CommandBuffer(u) uložit prokládané příkazy kreslení a čistění. K tomtu se váže test:

```
./izgProject -c --test 16
```

8.2.4 Test 17 - ověření ShaderInterface

Tento test zkouší, zda vertex shader obdržel správnou strukturu ShaderInterface. K tomtu se váže test:

```
./izgProject -c --test 17
```

ShaderInterface jsou konstantní vstupy do shaderu a vypadá následovně:

```
struct ShaderInterface{
  Uniform const*uniforms  = nullptr;
  Texture const*textures  = nullptr;
  uint32_t      gl_DrawID = 0      ;
};
```

Ukazatele ShaderInterface::uniforms a ShaderInterface::textures by měly obsahovat stejné ukazatele, které jsou uvedeny v paměti grafické karty GPUMemory::uniforms, GPUMemory::textures.

8.2.5 Test 18 - číslování vrcholů s indexováním.

Tento test zkouší využít indexační buffer pro číslování vrcholů InVertex::gl_VertexID

```
./izgProject -c --test 18
```

Musíte správně číslovat vstupní vrcholy, když je zapnuté indexování.

Indexování může být zapnuto nebo vypnuto - o tom rozhoduje nastavení ve struktuře VertexArray V paměti grafické karty je pole vertex array objektů GPUMemory::vertexArrays. Každý vertex array je tabulka nastavení takzvané vertex assembly jednotky (jednotka sestavující vrcholy). Struktura VertexArray vypadá následovně:

```
struct VertexArray{
  VertexAttrib vertexAttrib[maxAttribs];
  int32_t      indexBufferID = -1;
  uint64_t     indexOffset   = 0 ;
  IndexType    indexType     = IndexType::U32;
};
```

V této struktuře jsou pro indexovaní podstatné položky VertexArray::indexBufferID, VertexArray::indexOffset a VertexArray::indexType. indexBufferID je číslo bufferu nebo -1 pokud je indexing vypnutý. indexOffset je posun v bajtech od začátku bufferu, kde se nacházejí indexy. indexType je typ indexu.
Všechny buffery (stejně jako programy) se nachází v paměti grafické karty (GPUMemory).

```
struct GPUMemory{
  uint32_t         maxUniforms          = 0      ;
  uint32_t         maxVertexArrays      = 0      ;
  uint32_t         maxTextures          = 0      ;
  uint32_t         maxBuffers           = 0      ;
  uint32_t         maxPrograms          = 0      ;
  uint32_t         maxFramebuffers      = 0      ;
  uint32_t         defaultFramebuffer   = 0      ;
  Buffer          *buffers              = nullptr;
  Texture         *textures             = nullptr;
  Uniform         *uniforms             = nullptr;
  Program         *programs             = nullptr;
  Framebuffer     *framebuffers         = nullptr;
  VertexArray     *vertexArrays         = nullptr;
  uint32_t         activatedFramebuffer = 0      ;
  uint32_t         activatedProgram     = 0      ;
  uint32_t         activatedVertexArray = 0      ;
  uint32_t         gl_DrawID            = 0      ;
  StencilSettings  stencilSettings                ;
  BlockWrites      blockWrites                    ;
  BackfaceCulling  backfaceCulling                ;

  //Do not worry about these.
  //This is just to suppress valgrind warnings because of the large stack.
  //Otherwise everything would be placed on the stack and not on the heap.
  //I had to allocated this structure on the heap, because it is too large.
  GPUMemory();
  GPUMemory(GPUMemory const&o);
  ~GPUMemory();
  GPUMemory&operator=(GPUMemory const&o);
};
```

Buffer je lineární paměť, reprezentovano strukturou:

```
struct Buffer{
  void const* data = nullptr;
  uint64_t    size = 0      ;
};
```

Indexační buffer může mít různou velikost indexu - 8bit, 16bit a 32bit:

```
enum class IndexType : uint8_t{
  U8  = 1,
  U16 = 2,
  U32 = 4,
};
```

Pokud je zapnuto indexování, pak je číslo vrcholu dáno položkou v indexačním bufferu, kde je položka (index) v bufferu vybrána na základě čísla invokace vertex shaderu. Pseudokód:

```
uint32_t getIndex(GPUMemory const&mem,uint32_t i){
  // aktivovaná tabulka nastavení pro vertex assembly jednotu
  auto vao = mem.vertexArrays[mem.activatedVertexArray];
  if(vao.indexBufferID >= 0){ // je použito indexování?
    Buffer indexBuffer = mem.buffers[vao.indexBufferID]; //indexový buffer
    if(vao.indexType == IndexType::U8){ // je typ indexu uint8_t
      uint8_t*ptr = (uint8_t*)indexBuffer.data; //ukazatel na data
      ptr += vao.indexOffset; // posun ukazatele o offset v bajtech
      return ...
    }
  }else
    // ....
}
```

8.2.6 Testy 19-21 - Vertex Atributy, Vertex Assembly jednotka

Tyto testy ověřují, zda vám správně jednotka Vertex Assembly sestavuje vrcholy z paměti:

```
./izgProject -c --test 19
./izgProject -c --test 20
./izgProject -c --test 21
```

V tomto testu musíte naprogramovat funkcionalitu Vertex Assembly jednotky.

Vertex Assembly jednotka sestavuje vstupní vrcholy (InVertex) z paměti (GPUMemory) pomocí nastavení z tabulky VertexArray.

8.2.6.1 Vstupy:

Nastavení je uloženo ve struktuře VertexArray

```
struct VertexArray{
  VertexAttrib vertexAttrib[maxAttribs];
  int32_t      indexBufferID = -1;
  uint64_t     indexOffset   = 0 ;
  IndexType    indexType     = IndexType::U32;
};
```

Je složeno z nastavení pro indexování a nastavení pro vertex attributy.
VertexAttrib je struktura obsahují nastavení, jak číst jeden Vertex Attribut.

```
struct VertexAttrib{
  int32_t       bufferID = -1              ;
  uint64_t      stride   = 0               ;
  uint64_t      offset   = 0               ;
  AttribType    type     = AttribType::EMPTY;
};
```

Paměť GPUMemory obsahuje Buffer(y)

```
struct GPUMemory{
  uint32_t         maxUniforms          = 0      ;
  uint32_t         maxVertexArrays      = 0      ;
  uint32_t         maxTextures          = 0      ;
  uint32_t         maxBuffers           = 0      ;
  uint32_t         maxPrograms          = 0      ;
  uint32_t         maxFramebuffers      = 0      ;
  uint32_t         defaultFramebuffer   = 0      ;
  Buffer          *buffers              = nullptr;
  Texture         *textures             = nullptr;
  Uniform         *uniforms             = nullptr;
```

```
  Program         *programs             = nullptr;
  Framebuffer     *framebuffers         = nullptr;
  VertexArray     *vertexArrays         = nullptr;
  uint32_t         activatedFramebuffer = 0      ;
  uint32_t         activatedProgram     = 0      ;
  uint32_t         activatedVertexArray = 0      ;
  uint32_t         gl_DrawID            = 0      ;
  StencilSettings  stencilSettings                ;
  BlockWrites      blockWrites                    ;
  BackfaceCulling  backfaceCulling                ;

  //Do not worry about these.
  //This is just to suppress valgrind warnings because of the large stack.
  //Otherwise everything would be placed on the stack and not on the heap.
  //I had to allocated this structure on the heap, because it is too large.
  GPUMemory();
  GPUMemory(GPUMemory const&o);
  ~GPUMemory();
  GPUMemory&operator=(GPUMemory const&o);
};
```

Buffer je struktura obsahují pointer a velikost.

```
struct Buffer{
  void const* data = nullptr;
  uint64_t    size = 0      ;
};
```

8.2.6.2 Výstupy:

Strukura InVertex vypadá takto:

```
struct InVertex{
  Attrib   attributes[maxAttribs]    ;
  uint32_t gl_VertexID            = 0;
};
```

Data atributu vypadají takto:

```
union Attrib{
  Attrib():v4(glm::vec4(1.f)){}
  float      v1;
  glm::vec2  v2;
  glm::vec3  v3;
  glm::vec4  v4;
  uint32_t   u1;
  glm::uvec2 u2;
  glm::uvec3 u3;
  glm::uvec4 u4;
};
```

8.2.6.3 Úkol:

Vašim úkolem je správně číst data atributů z paměti a zapisovat je do struktury InVertex.

Po těchto úkolech byste měli mít hotovou vertexovou část. To je část, před sestavením primitiv (Primitiv Assembly Unit).

Chapter 9

08 Vektorová část GPU: část přimitiv

Vektorová část zobrazovacího řetězce lze rozdělit na dvě části:

- vertexová část,
- část primitiv.

Tyto dvě části jsou odděleny jednotkou Primitive Assembly, která ze streamu vertexů sestavuje stream primitiv.

Vertexovou čast byste v tuto chvíli již měli mít hotovou. Zbývá část primitiv.

9.1 Část primitiv

Vertex Assembly jednotka chrlí vrcholy a vertex shader je zpracovává, transformuje. Je na čase z nich sestavit trojúhelníky a připravit je pro rasterizaci. Část za vertex shaderem je složena z několika částí.

9.1.1 Primitive Assembly

Primitive Assembly je jednotka, která sestavuje trojúhelníky (mimo jiné). Trojúhelníku, úsečce, bodu se hromadně říká primitivum. V tomto projektu se používají pouze trojúhelníky. Primitive Assembly jednotka si počká na 3 po sobě jdoucí výstupní vrcholy z vertex shaderu a sestaví trojúhelník (struktura, která by měla obsahovat 3 výstupní vrcholy). Lze na to také nahlížet tak, že primitive assembly jednotka dostane příkaz vykreslit třeba 4 trojúhelníky. Jednotka tak spustí vertex shader 12x, který takto spustí 12x vertex assembly jednotku.

9.1.2 Perspektivní dělení

Perspektivní dělení následuje za clippingem (ten bude až později, teď není potřeba) a provádí převod z homogenních souřadnic na kartézské pomocí dělení w.

9.1.3 Viewport transformace

Viewport transformace provádí převod NDC (rozsah -1, +1) na rozlišení okna, aby se mohla provést rasterizace.

9.1.4 Culling / Backface Culling

Backface Culling se stará o zahození trojúhelníků, které jsou odvráceny od pozorovatele. Culling lze zapnout nebo vypnout pomocí: BackfaceCulling::enabled Pokud je zapnutý, trojúhelníky, které jsou odvrácené, jsou zahazovány. To, které jsou přivrácené a odvrácené je určeno:

- nastasvením BackfaceCulling::frontFaceIsCounterClockWise,
- pořadím vrcholů trojúhelníku na obrazovce, jsou-li specifikovány po směru nebo proti směru hodinových ručiček. Pokud je backface culling vypnutý, vykreslují se všechny trojúhelníky - přivrácené i odvrácené - specifikované po i proti směru hodinových ručiček.

Chapter 10

09 Rasterizace

V tomto úkolu je potřeba rozšířit funkcionalitu funkce student_GPU_run o schopnosti rasterizace. Cílem je naprogramovat části zobrazovacího řetězce, které jsou za vertex shaderem po rasterizaci a pouštění fragment shaderu (včetně). Vzhledem k tomu, že projekt nemůže automaticky testovat vektorovou část: část primitiv, jsou tyto testy odsunuty až k rasterizaci, kdy se jejich ověřování umožní.

10.1 Rasterizace

Rasterizace produkuje raterizací primitiva stream fragmentů:

Rasterizace rasterizuje primitiva v prostoru obrazovky (screen-space). Rasterizace produkuje fragmenty v případě, že **střed** pixelu leží uvnitř trojúhelníku.

10.2 InFragment

InFragment(y) odpovídají vzorkům v pixelu. Nesou hodnoty důležité pro výpočet jejich barvy.

Pozice InFragmentu InFragment::gl_FragCoord obsahuje 4 složky. Složka XY je souřadnice středu pixelu na obrazovce, Složka Z obsahuje hloubku. Poslední složka W není podstatná, ale obsahuje tím, čím se dělilo při perspektivním dělení.

10.3 Interpolace atributů

10.4 Výpočet 2D Barycentrických souřadnic pro interpolaci hloubky

Barycentrické souřadnice musíte spočítat podle obsahů: Hlouba se interpoluje pomocí barycentrických souřadnic ve 2D:

$$fragment.gl\_FragCoord.z = vertex[0].gl\_Position.z \cdot \lambda_0^{2D} + vertex[1].gl\_Position.z \cdot \lambda_1^{2D} + vertex[2].gl\_Position.z \cdot \lambda_2^{2D}$$

Hloubka vrcholů vertex[].gl_Position.z vznikla při perspektivním dělení.

10.5 Výpočet perspektivně korektních Barycentrických souřadnic pro interpolaci uživatelských atribů

Atributy je potřeba interpolovat pomocí perspektivně korektně upravených 2D barycentrických souřadnic. Perspektivně korektní interpolace:

$$\frac{\frac{A_0 \cdot \lambda_0^{2D}}{h_0} + \frac{A_1 \cdot \lambda_1^{2D}}{h_1} + \frac{A_2 \cdot \lambda_2^{2D}}{h_2}}{\frac{\lambda_0^{2D}}{h_0} + \frac{\lambda_1^{2D}}{h_1} + \frac{\lambda_2^{2D}}{h_2}}$$

Kde $\lambda_0^{2D}, \lambda_1^{2D}, \lambda_2^{2D}$ jsou barycentrické koordináty ve 2D, h_0, h_1, h_2 jsou homogenní složky vrcholů a A_0, A_1, A_2 jsou atribut vrcholu.
Homogenní složka vrcholů je čtvrtá složka - tím čím se dělilo ve perspektivním dělení: h0 = vertex[0].gl_Position.w, h1 = vertex[1].gl_Position.w, ...
2D Barycentrické souřadnice je možné přepočítat na perspektivně koretní barycentrické souřadnice (je to jen přepsání zvorečku nahoře):

$$s = \frac{\lambda_0^{2D}}{h_0} + \frac{\lambda_1^{2D}}{h_1} + \frac{\lambda_2^{2D}}{h_2}$$

$$\lambda_0 = \frac{\lambda_0^{2D}}{h_0 \cdot s}$$

$$\lambda_1 = \frac{\lambda_1^{2D}}{h_1 \cdot s}$$

$$\lambda_2 = \frac{\lambda_2^{2D}}{h_2 \cdot s}$$

Ty je potom možné použít pro interpolaci atributů:

$$fragment.attribute = vertex[0].attribute \cdot \lambda_0 + vertex[1].attribute \cdot \lambda_1 + vertex[2].attribute \cdot \lambda_2$$

10.6 Fragment processor

Fragment processor spouští fragment shader nad každým fragmentem. Data pro fragment shader jsou uložena ve struktuře InFragment. Výstup fragment shaderu je výstupní fragment OutFragment - barva. Další (konstantní) vstup fragment shaderu jsou uniformní proměnné a textury.

10.7 Úkol 4 - naprogramovat Primitive Assembly jednotku, perspektivní dělení, zahazování odvrácených primitiv, rasterizaci a pouštění fragment shaderu

Rasterizace rasterizuje primitiva. K tomu je potřeba korektně ty primitiva sestavit, provést perspektivní dělení a viewport transformaci. Jedná se o testy 22. - 31.

10.7.1 Test 22 - Ověření, že funguje základní rasterizace

V tomto úkolu budete muset naprogramovat rasterizaci. Neobejdete se bez viewport transformace, rasterizace a zavolání fragment shaderu nad každným fragmentem. Tento test spočívá ve zkoušení vyrasterizování jednoho trojúhelníku a podívání se, zda jste korektně pustili fragment shader.
Test spustíte:

```
izgProject -c --test 22
```

Pseudokód může po upravení vypadat nějak takto:

```
void vertexAssembly(){
  computeVertexID();
  readAttributes();
}

void primitiveAssembly(primitive,vertexArray,t,program){
  for(every vertex v in triangle){
    InVertex inVertex;
    vertexAssembly(inVertex,vertexArray,t+v);
    ShaderInterface si;
    if(program.vertexShader)
      program.vertexShader(primitive.vertex,inVertex,si);
  }
}

void rasterize(framebuffer,primitive,program){
  for(pixels in frame){
    if(pixels in primitive){
      InFragment inFragment;
      createFragment(inFragment,primitive,barycentrics,pixelCoord,program);
      OutFragment outFragment;
      ShaderInterface si;
      if(program.fragmentShader)
        program.fragmentShader(outFragment,inFragment,si);
    }
  }
}

void draw(GPUMemory&mem,DrawCommand const&cmd){
  for(every triangle t){
    Primitive primitive;
    primitiveAssembly(primitive,vertexArray,t,program)
    viewportTransformation(primitive,width,height)
    rasterize(framebuffer,primitive,program);
  }
}

void student_GPU_run(GPUMemory&mem,CommandBuffer const&cb){
  for(every command in cb){
    if(command.type == DRAW)
      draw(mem,command);
  }
}
```

10.7.2 Test 23 - Ověření, zda nerasterizujete mimo okno

V tomto testu jsou trojúhelníky částečně nebo zce mimo okno

```
izgProject -c --test 23
```

10.7.3 Test 24 - Komprehenzivní testování rasterizace

V tomto testu se testuje rasterizace mnoha trojúhelníků při mnoha nastaveních

```
izgProject -c --test 24
```

10.7.4 Test 25 - Ověření, zda počítáte perspektivní dělení.

Tento test ověřuje, zda provádíte perspektivní dělení.

```
izgProject -c --test 25
```

10.7.5 Test 26 - Ověření, zda vám funguje backface culling.

Tento test ověřuje, zda vám funguje backface culling.

```
izgProject -c --test 26
```

10.7.6 Test 27 - Ověření, zda se správně interpoluje hloubka fragmentů.

Tento test ověřuje, zda vyrasterizované fragmenty mají správně interpolovanou hloubku.

```
izgProject -c --test 27
```

Hloubka fragmentu je v komponentě "z" položky InFragment::gl_FragCoord. Pro její interpolaci potřebujete hloubky vrcholů trojúhelníka a barycentrické souřadnice fragmentu ve 2D.
Hloubky vrcholů najdete ve "z" komponentě položky OutVertex::gl_Position gl_Position.z

```
struct OutVertex{
  Attrib    attributes[maxAttribs]                    ;
  glm::vec4 gl_Position           = glm::vec4(0,0,0,1);
};
```

Hloubku zapisujte do komponenty "z" položky InFragment::gl_FragCoord gl_FragCoord.z

```
struct InFragment{
  Attrib    attributes[maxAttribs]              ;
  glm::vec4 gl_FragCoord          = glm::vec4(1);
};
```

10.7.7 Testy 28-29 - Ověření, zda se správně interpolují vertex attributy.

Tyto dva testy ověřují, jestli se správně interpolují vertex atributy do fragment atributů.

```
izgProject -c --test 28
izgProject -c --test 29
```

Vertex Attributy jsou se struktuře OutVertex

```
struct OutVertex{
  Attrib    attributes[maxAttribs]                    ;
  glm::vec4 gl_Position           = glm::vec4(0,0,0,1);
};
```

A ze tří těchto vrcholů by se měly interpolovat atributy InFragment.

```
struct InFragment{
  Attrib    attributes[maxAttribs]              ;
  glm::vec4 gl_FragCoord          = glm::vec4(1);
};
```

Interpolujte pouze ty atributy, které jsou poznačené v položce Program::vs2fs! A pouze ty, které nejsou typu integer! Integerové atributy neinterpolujte, ale pouze použijte hodnoty nultého vrcholu. Tomuto vrcholu se také říká provoking vertex.

```
struct Program{
  VertexShader   vertexShader   = nullptr;
  FragmentShader fragmentShader = nullptr;
  AttribType  vs2fs[maxAttribs] = {AttribType::EMPTY};
};
```

Chapter 11

10 Per Fragment Operace

11.1 Per Fragment Operace

Per Fragment operace jsou operace, které se provádí nad fragmenty, které vyprodukovala rasterizace. Jsou rozděleny na tři části:

- Brzké Per Fragment Operace
- Fragment Shader
- Pozdní Per Fragment Operace

Brzké Per Fragment Operace jsou složeny z:

- Stencil Testu
- Depth Testu

Pozdní Per Fragment Operace jsou složeny z:

- Operace Discard
- Zápisu stencilu
- Zápisu hloubky
- Zápisu barvy

Obrázek všech Per Fragment Operací (zjednodušené o reality, která je mnohem, mnohem složitější):

Chapter 12

12 Brzké per fragment operace

Mezi brzké Per Fragment Operace patří:

- stencilový test,
- hloubkový test.

12.1 Stencilový test

Stencilový test slouží k zahození fragmentů například pokud uživatel chce kreslit jen v části obrazovky. Využívá se například pro vykreslení portálů, zrcadel a podobných efektů. Ke stencilovému testu se váže stencilový buffer.

Nastavení stencilu je uloženo v paměti grafické karty GPUMemory::stencilSettings

```
struct StencilSettings{
  bool        enabled  = false             ;
  StencilFunc func     = StencilFunc::ALWAYS;
  uint32_t    refValue = 0                 ;
  StencilOps  frontOps                     ;
  StencilOps  backOps                      ;
};
```

12.1.1 Stencilová porovnávací funce

Pokud je stencilový test aktivní je rozhodnut na základě těchto věcí:

- Referenční hodnotě StencilSettings::refValue
- Stencilové funci StencilSettings::func
- Hodnotě ve stencilovému bufferu Framebuffer::stencil

Mezi stencilové funkce patří hodnoty z enumu StencilFunc

```
enum class StencilFunc{
  NEVER   ,
  LESS    ,
  LEQUAL  ,
  GREATER ,
  GEQUAL  ,
  EQUAL   ,
  NOTEQUAL,
  ALWAYS  ,
};
```

- StencilFunc::NEVER - nikdy neprojde
- StencilFunc::LESS - projde pokud je hodnota ve stencilovém buferu $<$ než StencilSettings::refValue
- StencilFunc::LEQUAL - projde pokud je hodnota ve stencilovém buferu $<=$ než StencilSettings::refValue
- StencilFunc::GREATER - projde pokud je hodnota ve stencilovém buferu $>$ než StencilSettings::refValue
- StencilFunc::GEQUAL - projde pokud je hodnota ve stencilovém buferu $>=$ než StencilSettings::refValue
- StencilFunc::EQUAL - projde pokud je hodnota ve stencilovém buferu == než StencilSettings::refValue
- StencilFunc::NOTEQUAL - projde pokud je hodnota ve stencilovém buferu != než StencilSettings::refValue
- StencilFunc::NEVER - vždy projde

12.1.2 Stencilová operace

Stencilová operace závisí na tom, zda je rasterizován přivrácený nebo odvrácený trojúhelník. Pokud se rasterizuje přivrácený trojúhelník, měla by se použít operace z StencilSettings::frontOps jinak StencilSettings::backOps.

Mezi stencilové opereace patří hodnoty z enumu StencilOp

```
enum class StencilOp{
   KEEP     ,
   ZERO     ,
   REPLACE  ,
   INCR     ,
   INCR_WRAP,
   DECR     ,
   DECR_WRAP,
   INVERT   ,
};
```

- StencilOp::KEEP - ponechá hodnotu ve stencilovém bufferu nezměněnou
- StencilOp::ZERO - vynuluje hodnotu ve stencilovém bufferu
- StencilOp::REPLACE - zapíše do stencil bufferu referenční hodnotu StencilSettings::refValue
- StencilOp::INCR - inkrementuje hodnotu ve stencilovém bufferu, pokud je 255, nechá ji být
- StencilOp::INCR_WRAP - inkrementuje hodnotu ve stencilovému bufferu, pokud je 255, zapíše 0
- StencilOp::DECR - dekrementuje hodnotu ve stencilovém bufferu, pokud je 0, nechá ji být
- StencilOp::DECR_WRAP - dekrementuje hodnotu ve stencilovém bufferu, pokud je 0, zapíše 255
- StencilOp::INVERT - bitově invertuje hodnotu ve stencilovém bufferu

Pokud stencil test neprojde a je možné modifikovat stencilový buffer, modifikuje se na základě operace StencilOps::sfail.

```
struct StencilOps{
  StencilOp sfail  = StencilOp::KEEP;
  StencilOp dpfail = StencilOp::KEEP;
  StencilOp dppass = StencilOp::KEEP;
};
```

12.2 Hloubkový test

Hloubkový test je druhá z brzkých per fragment operací. Stará se o zahazování fragmentů, které jsou hlouběji než to, co už se vyrasterizovalo. Využívá k tomu hloubkový buffer.

Hloubka fragmentu je "z" komponenta InFragment::gl_FragCoord InFragment::gl_FragCoord.z. Pokud je hloubka nového fragment menší, hloubkový test prošel.

12.2.1 Pokud hloubkový test selže...

Podobně jako je tomu, když selže stencilový test, pokud selže hloubkový test, je možné, že se provede stencilová operace. Tentokrát však to bude operace StencilOps::dpfail.

12.3 Úkol 5 - Naprogramovat brzké Per Fragment Operace

12.3.1 Test 30 - Stencil test

Tento test zkouší, zda funguje stencilový test
```
./izgProject -c --test 30
```

12.3.2 Test 31 - Stencil Operace při sfail

Tento test zkouší, zda fungují stencilové operace, když stencilový test selže
```
./izgProject -c --test 31
```

12.3.3 Test 32 - Depth test

Tento test zkouší, zda funguje depth test.
```
./izgProject -c --test 32
```

12.3.4 Test 33 - Depth test a modifikace stencilového buffer při dpfail

Tento test zkouší, zda fungují modifikace stencilového bufferu při dpfail.
```
./izgProject -c --test 33
```

Chapter 13

13 Pozdní Per Fragment Operace

Pozdní Per Fragment Operace se nachází za fragment shaderem. Jsou složeny z:

- Zahazování fragmentů (operace discard)
- Možné modifikace stencilového bufferu (stencil operace dppass)
- Možné modifikace hloubkového bufferu (pokud je depth buffer přítomen)
- Možného zápisu barvy do barevného bufferu (pokud je barevný buffer přítomen)

13.1 Zahazování fragmentu (operace discard)

Zahazování fragmentů je určeno na základě OutFragment::discard. Pokud je nastavena na true, je fragment zahozen a jeho další zpracovávání je ukončeno.

13.2 Modifikace stencilového bufferu

Zápis do stencilového bufferu je umožněn ještě jednou, naposled. Pokud je zápis povolen a pokud je stencil buffer přítomen, provede se nad ním operace StencilOps::dppass.

13.3 Modifikace hloubkového bufferu

Zápis hloubky zajistí, že budoucí fragmenty nepřepíší barvu bližších objektů. Zápis je povolen v případě, že je přítomen hloubkový buffer.

13.4 Modifikace barevného bufferu

Zápis barvy do barevného bufferu je umožněn, pokud je barevný buffer přítomen.

Pokud se má zapsat barva, je zapsána pomocí takzvaného Blendingu. Blending využívá průhlednosti fragmentu k přímíchání nové barvy ke stávající v color bufferu. Blending má v reálu mnoho nastavení, v projetku se používá pouze alpha blending. Fragmenty mají barvu RGBA, kde A - α je tzv. neprůhlednost.
Pokud má nový fragment $\alpha = 1$ - je absolutně neprůhledný - plně přepíše barvu ve framebufferu.
Pokud má nový fragment $\alpha = 0$ - je absolutně průhledný - vůbec barvu ve framebuffer nezmění.
Pokud má hodnotu někde mezi, tak se barva lineárně smíchá:

$$colorBuffer_{rgb} = colorBuffer_{rgb} \cdot (1 - \alpha) + gl\_FragColor_{rgb} \cdot \alpha$$

Kde $\alpha = gl\_FragColor_a$

13.5 Úkol 6 - Naprogramovat pozdní Per Fragment Operace

13.5.1 Test 34 - Discard

Tento test zkouší, zda funguje operace discard.
```
./izgProject -c --test 34
```

13.5.2 Test 35 - Modifikace stencilového bufferu při dppass

Tento test zkouší, zda funguje stencilová operace při dppass
```
./izgProject -c --test 35
```

13.5.3 Test 36 - Modifikace depth bufferu

Tento test zkouší, zda funguje zápis do hloubkového bufferu.
```
./izgProject -c --test 36
```

13.5.4 Test 37 - Zápis barvy a blending

Tento test zkouší, zda funguje zápis do barevného bufferu.
```
./izgProject -c --test 37
```

Chapter 14

14 Ořez

Tento úkol opravuje vykreslování pokud je geometrie za pozorovatelem. Tyto úkoly můžete přeskočit a vrátit se k nim později. Pokud se na geometrii budete dívat tak, že leží vždy před vámi, nepoznáte rozdíl.

14.1 Teorie ořezu

Ořez (clipping) slouží pro odstranění částí trojúhelníků, které leží mimo pohledový jehlan. Nejdůležitější je však ořez near ořezovou rovinou pohledoveho jehlanu. Pokud by se neprovedl ořez pomocí near roviny, pak by se vrcholy nebo i celé trojúhelníky, které leží za středem projekce promítly při perspektivním dělení na průmětnu. Ořez se provádí v clip-space - po Primitive Assembly jednotce. Pro body, které leží uvnitř pohledového tělesa platí, že jejich souřadnice splňují následující nerovnice: $-A_w \leq A_i \leq +A_w$, $i \in \{x, y, z\}$. Těchto 6 nerovnic reprezentuje jednotlivé svěny pohledového jehlanu. Nerovnice $-A_w \leq A_z$ reprezentuje podmínku pro near ořezovou rovinu.
Při ořezu trojúhelníku můžou nastat 4 případy, jsou znázorněny na následujícím obrázku:

Ořez trojúhelníku pomocí near roviny lze zjednodušit na ořez hran trojúhelníku. Bod na hraně (úsečce) trojúhelníku lze vyjádřit jako: $\overrightarrow{X(t)} = \overrightarrow{A} + t \cdot (\overrightarrow{B} - \overrightarrow{A})$, $t \in [0, 1]$. $\overrightarrow{A}$, $\overrightarrow{B}$ jsou vrcholy trojúhelníka, $\overrightarrow{X(t)}$ je bod na hraně a parametr t udává posun na úsečce.

Souřadnice bodu $\overrightarrow{X(t)}$ lze určit při vypočtení parametru t, při kterém přestane platit nerovnice pro near rovinu $-X(t)_w \leq X(t)_z$. Takové místo nastává v situaci $-X(t)_w = X(t)_z$. Po dosazení z rovnice úsečky lze vztah přepsat na:

$$\begin{aligned}
-X(t)_w &= X(t)_z \\
0 &= X(t)_w + X(t)_z \\
0 &= A_w + t \cdot (B_w - A_w) + A_z + t \cdot (B_z - A_z) \\
0 &= A_w + A_z + t \cdot (B_w - A_w + B_z - A_z) \\
-A_w - A_z &= t \cdot (B_w - A_w + B_z - A_z) \\
\frac{-A_w - A_z}{B_w - A_w + B_z - A_z} &= t
\end{aligned}$$

Pozice bodu $\overrightarrow{X(t)}$ a hodnoty dalších vertex atributů lze vypočítat lineární kombinací hodnot z vrcholů úsečky pomocí parametru t následovně: $\overrightarrow{X(t)} = \overrightarrow{A} + t \cdot (\overrightarrow{B} - \overrightarrow{A})$.

14.2 Úkol 7 - naprogramovat ořez trojúhelníků blízkou ořezovou rovinou

Testy, které kontrolují ořez, jsou 38. - 41:
```
izgProject -c --test 41 --up-to-test 41
```

Pseudokód ořezu může vypadat takto:
```
void draw(mem,drawCommand){
  for(every triangle t){
    Primitive primitive;
    runPrimitiveAssembly(primitive,vertexArray,t,vertexShader)

    ClippedPrimitive clipped;
    performeClipping(clipped,primitive);

    for(all clipped triangle c in clipped){
      runPerspectiveDivision(c)
      runViewportTransformation(c,width,height)
      rasterizeTriangle(framebuffer,c,fragmentShader);
    }
  }
}

void student_GPU_run(mem,commandBuffer){
  for(every command in commandBuffer){
    if(isDrawCommand )draw (mem,drawCommand)
  }
}
```

14.2.1 Test 38 - ořez celého CW trojúhelníku, který je příliš blízko kamery.

Tento test zkouší, zda funguje ořez celého trojúhelníka definovaného jako CC (clock wise).
```
izgProject -c --test 38
```

14.2.2 Test 39 - ořez celého CCW trojúhelníku, který je příliš blízko kamery.

Tento test zkouší, zda funguje ořez celého trojúhelníku definovaného jako CCW (counter clock wise)
```
izgProject -c --test 39
```

14.2.3 Test 40 - Ořez trojúhelníku, když je 1 vrchol ořezán

Tento test zkouší ořezat trojúhelník, když je mimo pohledové těleso jeden vrchol.
```
izgProject -c --test 40
```

14.2.4 Test 41 - Ořez trojúhelníku, když jsou 2 vrcholy ořezány

Tento test zkouší ořezat trojúhelník, když jsou mimo pohledové těloso dva vrcholy.
```
izgProject -c --test 41
```

14.3 Hotová grafická karta

Pokud budete mít ořez hotový, dokončili jste implementaci grafické karty! Byla to fuška, ale věřte, že skutečné grafické karty jsou alespoň milionkrát složitější. Takto vypadá celý vykreslovací řetězec:

Měly by vám fungovat příklady, které nevyžadují načítání modelů: Další úkoly jsou zaměřené už na vykreslování modelů s využitím stínů.

Chapter 15

15 Implementace vykreslování modelů se stíny - soubor student/prepareModel.cpp

Druhá věc, co se asi ptáte je: "K čemu se dá grafická karta využít?" Cílem této části projektu je vykreslit modely se stíny pomocí vámi vytvořené grafické karty. Všechny úkoly této části se týkají souboru student/student_prepare↩ Model.cpp.

15.1 Úkol 8 - Vykreslování modelů - funkce student_prepareModel

Tento úkol už se neváže k zobrazovacímu řetězci, ale k jeho využívání. Cílem je naprogramovat zobrazování modelů načtených ze souboru na disku. Načítání modelů už je uděláno a předpřipraveno. Vaším úkolem je jen správně vytvořit command buffer a zapsat správně data do grafické karty. Budete editovat funkci student_prepareModel v souboru studentSolution/src/studentSolution/prepareModel.cpp. Samotné volání kreslení nebudete dělat, připravujete command buffer a paměť, které zpracuje příklad modelMethod.cpp.
Struktura modelu je: Vážou s k němu struktury Model, Node, Mesh, Buffer, Texture.

```
struct Model{
  Node*   roots       = nullptr;
  Buffer* buffers     = nullptr;
  Mesh*   meshes      = nullptr;
  Texture*textures    = nullptr;
  size_t  nofRoots    = 0      ;
  size_t  nofBuffers  = 0      ;
  size_t  nofMeshes   = 0      ;
  size_t  nofTextures = 0      ;
};
struct Node{
  glm::mat4 modelMatrix = glm::mat4(1.f);
  int32_t   mesh        = -1            ;
  Node*     children    = nullptr       ;
  size_t    nofChildren = 0             ;
};
struct Mesh{
  int32_t      indexBufferID  = -1            ;
  size_t       indexOffset    = 0             ;
  IndexType    indexType      = IndexType::U32;
  VertexAttrib position                       ;
  VertexAttrib normal                         ;
  VertexAttrib texCoord                       ;
  uint32_t     nofIndices     = 0             ;
  glm::vec4    diffuseColor   = glm::vec4(1.f);
  int          diffuseTexture = -1            ;
  bool         doubleSided    = false         ;
};
struct Buffer{
  void const* data = nullptr;
  uint64_t    size = 0      ;
};
struct Texture{
  uint32_t width  = 0;
  uint32_t height = 0;
```

```
  Image    img       ;
};
```

Pro správné vytvoření command bufferu je potřeba projít kořeny modelu a vložit všechny uzly, které mají mesh. Procházejte stromy průchodem pre order. Uzly se mohou odkazovat na mesh nebo nemusí (pokud je mesh=-1).
Mesh se může odkazovat na texturu nebo nemusí (pokud je diffuseTexture=-1).
V zásadě jde o to ke každému uzlu, ve kterém je odkaz na mesh, vytvořit DrawCommand a vložit jej do CommandBuffer a vytvořit VertexArray a vložit jej do paměti grafické karty GPUMemory.
Je potřeba správně spočítat modelové matice, které se budují postupný pronásobováním z kořenového uzlu.
Vytvoření command bufferu lze napsat s výhodou rekurzivně. Pseudokód možné implementace:

```
void student_prepareNode(GPUMemory&mem,CommandBuffer&cb,Node const&node,Model const&model,glm::mat4
      const&prubeznaMatice,...){
  if(node.mesh>=0){
    Mesh mesh = model.meshes[node.mesh];

    drawCounter; // pocitadlo kreslicich prikazu

    // vytvoření vertex array
    VertexArray vao;
    vao.indexBufferID = mesh.indexBufferID;
    vao.indexOffset = ...;
    vao.indexType = ...;
    vao.vertexAttrib[0] = ...; // pozice
    vao.vertexAttrib[1] = ...; // normala
    vao.vertexAttrib[2] = ...; // texturovaci souradnice

    // vlozeni vao na spravne misto v pameti (aby jej bylo mozne najit
    mem.vertexArrays[drawCounter] = vao;

    BindVertexArrayCommand bindVaoCmd;
    bindVaoCmd.id = drawCounter;

    DrawCommand drawCmd;
    drawCmd.backfaceCulling = ...; // pokud je double sided tak by se nemelo orezavat
    drawCmd.nofVertices = ...; // pocet vertexu

    // vlozeni bindVaoCmd a drawCmd do command buffer cb
    cb.commands[...    ] = bindVaoCmd;
    cb.commands[... +1 ] = setBackfaceCulling;
    cb.commands[... +2 ] = drawCmd   ;

    //zapis uniformnich dat do pameti
    ZKOBINUJ(prubeznaMatice,node.modelMatrix);
    vypocitej inverzni transponovanou matici pro normaly...

    mem.uniforms[getUniformLocation(cmdID,MODEL_MATRIX                  )].m4 = modelMatrix
      ;
    mem.uniforms[getUniformLocation(cmdID,INVERSE_TRANSPOSE_MODEL_MATRIX)].m4 = inverzniTransponovana
      Modelova;
    mem.uniforms[getUniformLocation(cmdID,DIFFUSE_COLOR                 )].v4 = difuzni barva
      ;
    mem.uniforms[getUniformLocation(cmdID,TEXTURE_ID                    )].i1 = id textury nebo -1 pokud
      neni ;
    mem.uniforms[getUniformLocation(cmdID,DOUBLE_SIDED                  )].v1 = double sided
      ;



    writeToMemory(mem);
  }

  for(size_t i=0;i<node.children.size();++i)
    prepareNode(mem,node.children[i],model,...); rekurze
}

void prepareModel(GPUMemory&mem,CommandBuffer&cb,Model const&model){
  mem.buffers  = ...;
  mem.textures = ...;

  glm::mat4 jednotkovaMatrice = glm::mat4(1.f);
  for(size_t i=0;i<model.roots.size();++i)
    prepareNode(mem,cb,model.roots[i],jednotkovaMatrice,...);
}
```

Příklad, jak připravit command buffer, můžete najít v souboru examples/phongMethod.cpp

```
void vertexShader(OutVertex&outVertex,InVertex const&inVertex,ShaderInterface const&si){
  auto const pos = glm::vec4(inVertex.attributes[0].v3,1.f);
  auto const&nor = inVertex.attributes[1].v3;
  auto const&viewMatrix       = si.uniforms[0].m4;
```

```
  auto const&projectionMatrix = si.uniforms[1].m4;

  auto mvp = projectionMatrix*viewMatrix;

  outVertex.gl_Position = mvp * pos;
  outVertex.attributes[0].v3 = pos;
  outVertex.attributes[1].v3 = nor;
}

void fragmentShader(OutFragment&outFragment,InFragment const&inFragment,ShaderInterface const&si){
  auto const& light          = si.uniforms[2].v3;
  auto const& cameraPosition = si.uniforms[3].v3;
  auto const& vpos           = inFragment.attributes[0].v3;
  auto const& vnor           = inFragment.attributes[1].v3;
  auto vvnor = glm::normalize(vnor);

  auto l = glm::normalize(light-vpos);
  float diffuseFactor                    = glm::dot(l, vvnor);
  if (diffuseFactor < 0.f) diffuseFactor = 0.f;

  auto v = glm::normalize(cameraPosition-vpos);
  auto r = -glm::reflect(v,vvnor);
  float specularFactor                     = glm::dot(r, l);
  if (specularFactor < 0.f) specularFactor = 0.f;
  float const shininess                    = 40.f;

  if (diffuseFactor < 0)
    specularFactor = 0;
  else
    specularFactor = powf(specularFactor, shininess);

  float t = vvnor[1];
  if(t<0.f)t=0.f;
  t*=t;
  auto materialDiffuseColor = glm::mix(glm::vec3(0.f,1.f,0.f),glm::vec3(1.f,1.f,1.f),t);

  float const nofStripes = 10;
  float factor = 1.f / nofStripes * 2.f;

  auto xs = static_cast<float>(glm::mod(vpos.x+glm::sin(vpos.y*10.f)*.1f,factor)/factor > 0.5);

  materialDiffuseColor =
      glm::mix(glm::mix(glm::vec3(0.f,.5f,0.f),glm::vec3(1.f,1.f,0.f),xs),glm::vec3(1.f),t);

  auto materialSpecularColor = glm::vec3(1.f);

  auto diffuseColor  = materialDiffuseColor  * diffuseFactor;
  auto specularColor = materialSpecularColor * specularFactor;

  auto const color = glm::min(diffuseColor + specularColor,glm::vec3(1.f));
  outFragment.gl_FragColor = glm::vec4(color,1.f);
}

Method::Method(GPUMemory&m,MethodConstructionData const*): ::Method(m){
  mem.buffers[0].data = (void const*)bunnyVertices;
  mem.buffers[0].size = sizeof(bunnyVertices);
  mem.buffers[1].data = (void const*)bunnyIndices;
  mem.buffers[1].size = sizeof(bunnyIndices);
  mem.programs[0].vertexShader   = vertexShader;
  mem.programs[0].fragmentShader = fragmentShader;
  mem.programs[0].vs2fs[0]       = AttribType::VEC3;
  mem.programs[0].vs2fs[1]       = AttribType::VEC3;

  mem.vertexArrays[0].vertexAttrib[0].bufferID   = 0                 ;
  mem.vertexArrays[0].vertexAttrib[0].type       = AttribType::VEC3;
  mem.vertexArrays[0].vertexAttrib[0].stride     = sizeof(BunnyVertex);
  mem.vertexArrays[0].vertexAttrib[0].offset     = 0                 ;
  mem.vertexArrays[0].vertexAttrib[1].bufferID   = 0                 ;
  mem.vertexArrays[0].vertexAttrib[1].type       = AttribType::VEC3;
  mem.vertexArrays[0].vertexAttrib[1].stride     = sizeof(BunnyVertex);
  mem.vertexArrays[0].vertexAttrib[1].offset     = sizeof(glm::vec3)  ;
  mem.vertexArrays[0].indexBufferID = 1               ;
  mem.vertexArrays[0].indexOffset   = 0               ;
  mem.vertexArrays[0].indexType     = IndexType::U32;

  pushClearColorCommand(commandBuffer,glm::vec4(.5,.5,.5,1));
  pushClearDepthCommand(commandBuffer,10e10f);
  pushBindProgramCommand(commandBuffer,0);
  pushBindVertexArrayCommand(commandBuffer,0);
  pushDrawCommand (commandBuffer,sizeof(bunnyIndices)/sizeof(VertexIndex));
}


void Method::onDraw(SceneParam const&sceneParam){
  mem.uniforms[0].m4 = sceneParam.view  ;
  mem.uniforms[1].m4 = sceneParam.proj  ;
  mem.uniforms[2].v3 = sceneParam.light ;
```

```
  mem.uniforms[3].v3 = sceneParam.camera;

  gpuRun(mem,commandBuffer);
}
```

K tomuto úkolu se vážou testy 42. až 55
```
./izgProject -c --test 42
./izgProject -c --test 43
...
./izgProject -c --test 55
```

15.1.1 Testy 42-47 - Průchod modelem

Testy 42. - 47. kontrolují, jestli správně vytváříte command buffer.

15.1.2 Testy 48-55 - paměť

Testy 48. - 55. kontrolují, jestli správně plníte paměť grafické karty.

15.2 Úkol 9 - Vykreslování modelů - vertex shader student_drawModel_vertexShader

Funkce student_drawModel_vertexShader reprezentuje vertex shader pro zobrazení modelů.
Jeho funkcionalita spočívá v transformování vrcholů pomocí matic.
Vstupem jsou vrcholy, které mají pozici (3f), normálu (3f) a texturovací souřadnice (2f) (atributy 0, 1 a 2).
Vertex Attributy InVertex:

- inVertex.attributes[0].v3 - pozice vertexu v model-space
- inVertex.attributes[1].v3 - normála vertexu v model-space
- inVertex.attributes[2].v2 - tex. koordináty

Výstupem jsou vrcholy, které mají pozici (3f) a normálu (3f) ve world space, texturovací souřadnice (2f) a pozici vrcholu v clip-space světla (4f) (atributy 0, 1, 2, 3).
Vertex Attributy OutVertex:

- outVertex.attributes[0].v3 - pozice vertexu ve world-space
- outVertex.attributes[1].v3 - normála vertexu ve world-space
- outVertex.attributes[2].v2 - tex. koordináty
- outVertex.attributes[3].v4 - pozice vertexu v clip-space světla.

Uniformní proměnné obsahují projectionView matici, modelovou matici, a inverzní transponovanou matici.
Uniformní proměnné Uniforms:

- si.uniforms[getUniformLocation(gl_DrawID,PROJECTION_VIEW_MATRIX)].m4 - cameraProjectionView projekční a view matice kamery

- si.uniforms[getUniformLocation(gl_DrawID,USE_SHADOW_MAP_MATRIX)].m4 - lightProjectionView projekční a view matice světla - pro stíny
- si.uniforms[getUniformLocation(gl_DrawID,MODEL_MATRIX)].m4 - modelová matice
- si.uniforms[getUniformLocation(gl_DrawID,INVERSE_TRANSPOSE_MODEL_MATRIX)].m4 - inverzní transponovaná matice
- s.gl_DrawID - číslo vykreslovacího příkazu

Pozice by se měla pronásobit modelovou maticí "m∗glm::vec4(pos,1.f)", aby se ztransformovala do world-space.
Normála by se měla pronásobit inverzní transponovanou modelovou maticí "itm∗glm::vec4(nor,0.f)" aby se dostala do world-space.
Texturovací souřadnice se pouze přepošlou.
Pozice vrcholu gl_Position by měla být vypočtena pronásobením cameraProjectionView∗model∗pos.
Pozice vrcholu v prostoru clip-space prostoru světla pro stíny by se měla vypočítat lightProjectionView∗model∗pos.
K tomuto úkolu se váže tests 56.

15.2.1 Test 56 - Vertex Shader

```
izgProject -c --test 56
```

15.3 Úkol 10 - Vykreslování modelů - fragment shader drawMode_fragmentShader

Funkce student_drawModel_fragmentShader reprezentuje fragment shader pro zobrazení modelů.
Jeho funkcionalita spočívá v obarvování fragmentů, počítání lambertova osvětlovacího modelu a výpočtu stínu.
Vstupem jsou fragmenty, které mají: pozici (3f), normálu (3f), texturovací souřadnice (2f) a pozici v clip-space prostoru světla pro čtení ze stínové mapy. (atributy 0,1,2,3).
Fragment Attributy InFragment:

- inFragment.attributes[0].v3 - pozice fragmentu ve world-space
- inFragment.attributes[1].v3 - normála fragmentu ve world-space
- inFragment.attributes[2].v2 - tex. koordináty
- inFragment.attributes[3].v4 - pozice fragmentu v clip-space světla pro adresaci stínové mapy a výpočet stínu

Výstupem je fragment s barvou a správnou průhledností α.

Uniformní proměnné obsahují pozici světla (3f), pozici kamery (3f), difuzní barvu (4f), číslo textury (1i) a příznak doubleSided (1f).
Vzhledem k tomu, že má každý mesh jinou texturu a jiné nastavení, je nutné najít správné textury podle gl_DrawID.
Uniformní proměnné Uniforms:

- si.uniforms[getUniformLocation(gl_DrawID,LIGHT_POSITION)].v3 - pozice světla ve world-space
- si.uniforms[getUniformLocation(gl_DrawID,CAMERA_POSITION)].v3 - pozice kamery ve world-space
- si.uniforms[getUniformLocation(gl_DrawID,SHADOWMAP_ID)].i1 - číslo textury, která obsahuje stínovou mapu, nebo -1 pokud stíny nejsou
- si.uniforms[getUniformLocation(gl_DrawID,AMBIENT_LIGHT_COLOR)].v3 - barva ambientního světla
- si.uniforms[getUniformLocation(gl_DrawID,LIGHT_COLOR)].v3 - barva světla

- si.uniforms[getUniformLocation(gl_DrawID,DIFFUSE_COLOR)].v4 - difuzní barva
- si.uniforms[getUniformLocation(gl_DrawID,TEXTURE_ID)].i1 - číslo textury nebo -1 pokud textura není
- si.uniforms[getUniformLocation(gl_DrawID,DOUBLE_SIDED)].v1 - příznak doubleSided (1.f pokud je, 0.↩ f pokud není)

Vstupní normálu byste měli znormalizovat N=glm::normalize(nor).
Difuzní barva materiálu je buď uložena v uniformní proměnné nebo v textuře.
Rozhoduje se podle toho, jestli je čístlo textury záporné nebo ne.
Pokud je nastaven příznak doubleSided (je $>$ 0), jedná se o doustraný povrch.
V takovém případě je nutné otočit normálu, pokud je otočená od kamery (využijte pozici kamery v uniformní proměnné).
Spočítejte lambertův osvětlovací model se stíny pomocí shadow mappingu.
Spočítejte, zda je fragment ve stínu.
K tomu je potřeba vyčíst hloubku ze stínové mapy a porovnat ji se vzdáleností ke světlu.
Testy vás povedou.
K tomuto úkolu se váže testy 57-61.

15.3.1 Test 57-61 - Fragment Shader

```
izgProject -c --test 57
izgProject -c --test 58
izgProject -c --test 59
izgProject -c --test 60
izgProject -c --test 61
```

Ukázka, jak se počítá celý shadow mapping je v souboru:

```
class Method: public ::Method{
  public:
    Method(GPUMemory&m,MethodConstructionData const*);
    virtual ~Method(){}
    virtual void onDraw(SceneParam const&sceneParam) override;
    virtual void onUpdate(float dt) override;
    float time = 0;
    CommandBuffer commandBuffer;
    TextureData shadowMap;
};

// struktura reprezentujici vertex
struct Vertex{
  vec3 pos; // pozice
  vec3 col; // barva
};

// vertex sceny - dva čtverce, jeden zeleny a druhy červený
Vertex const vertices[] = {
  {vec3(-8,8,-8),vec3(1,0,0)},
  {vec3(+8,8,-8),vec3(1,0,0)},
  {vec3(-8,8,+8),vec3(1,0,0)},
  {vec3(-8,8,+8),vec3(1,0,0)},
  {vec3(+8,8,-8),vec3(1,0,0)},
  {vec3(+8,8,+8),vec3(1,0,0)},

  {vec3(-100,0,-100),vec3(0,1,0)},
  {vec3(+100,0,-100),vec3(0,1,0)},
  {vec3(-100,0,+100),vec3(0,1,0)},
  {vec3(-100,0,+100),vec3(0,1,0)},
  {vec3(+100,0,-100),vec3(0,1,0)},
  {vec3(+100,0,+100),vec3(0,1,0)},

};

// vertex shader pro vytvoření shadow mapy
void createShadowMap_vs(OutVertex&outVertex,InVertex const&inVertex,ShaderInterface const&si){
  auto gl_VertexID = inVertex.gl_VertexID;

  // light view matice
  auto view = si.uniforms[2].m4;
  // light projekční matice
  auto proj = si.uniforms[3].m4;

  // vypočet pozice vrcholu v clip-space
  outVertex.gl_Position = view*vec4(vertices[gl_VertexID].pos,1.f);
```

```
  outVertex.gl_Position.z -= .5f; // bias (proti self shadowingu)
  outVertex.gl_Position = proj * outVertex.gl_Position;
}

// nepotřebujeme fragment shader, stačí nám hloubka
void createShadowMap_fs(OutFragment&,InFragment const&,ShaderInterface const&){
}

// vertex shader pro výpočet stínu
void scene_vs(OutVertex&outVertex,InVertex const&inVertex,ShaderInterface const&si){
  // číslo vrcholu
  auto gl_VertexID = inVertex.gl_VertexID;

  // view matice kamery
  auto view = si.uniforms[0].m4;
  // projekční matice kamery
  auto proj = si.uniforms[1].m4;
  // view matice světla
  auto lightView = si.uniforms[2].m4;
  // projekční matice světla
  auto lightProj = si.uniforms[3].m4;
  // bias matice světla
  auto lightBias = si.uniforms[4].m4;

  // pozice vertexu ve world-space
  auto vertex = vec4(vertices[gl_VertexID].pos,1.f);

  // zápis barvy
  outVertex.attributes[0].v3 = vertices[gl_VertexID].col;
  // zápis pozice vertexu v clip-space světla, tady jsou uvedeny všechny matice explicitně
  outVertex.attributes[1].v4 = lightBias*lightProj*lightView*vertex;
  // zápis pozice vertexu v clip-space kamery
  outVertex.gl_Position = proj*view*vertex;
}

// fragment shader pro výpočet stínu
void scene_fs(OutFragment&outFragment,InFragment const&inFragment,ShaderInterface const&si){
  // barva
  auto color     = inFragment.attributes[0].v3;
  // pozice fragmentu v clip-space světla
  auto shadowPos = inFragment.attributes[1].v4;

  // perspektivní dělení
  shadowPos/=shadowPos.w;

  // vyčtení hloubky ze stínové mapy
  auto sm = read_textureClamp(si.textures[1],glm::vec2(shadowPos)).r;

  // je hloubka fragmentu větší než to, co je ve stínové mapě?
  auto isShadow = (float)(shadowPos.z > sm);

  // útlum barvy stínem
  color *= (1.f - .5f*isShadow);

  // zápis barvy
  outFragment.gl_FragColor = vec4(color,1.f);
}


Method::Method(GPUMemory&m,MethodConstructionData const*): ::Method(m){
  // vytvoření stínové mapy (data)
  shadowMap = TextureData(m.framebuffers[0].width,m.framebuffers[0].height,1,Image::F32);

  // program pro vytvoření stínové mapy
  auto&prg0 = m.programs[0];
  prg0.vertexShader   = createShadowMap_vs;
  prg0.fragmentShader = createShadowMap_fs;

  // program pro vykreslení scény se stíny
  auto&prg1 = m.programs[1];
  prg1.vertexShader   = scene_vs;
  prg1.fragmentShader = scene_fs;
  prg1.vs2fs[0] = AttribType::VEC3;
  prg1.vs2fs[1] = AttribType::VEC4;

  // framebuffer pro vykreslování stínové mapy
  m.textures[1] = shadowMap.getTexture();
  m.framebuffers[1].depth  = m.textures[1].img;
  m.framebuffers[1].width  = m.textures[1].width;
  m.framebuffers[1].height = m.textures[1].height;

  // vykreslení stínové mapy
  pushBindFramebufferCommand(commandBuffer,1);
  pushBindProgramCommand(commandBuffer,0);
  pushClearColorCommand(commandBuffer,glm::vec4(0,0,0,1));
  pushClearDepthCommand(commandBuffer);
```

```
  pushDrawCommand(commandBuffer,12);

  // vykreslení scény
  pushBindFramebufferCommand(commandBuffer,0);
  pushBindProgramCommand(commandBuffer,1);
  pushClearColorCommand(commandBuffer,glm::vec4(0,0,0,1));
  pushClearDepthCommand(commandBuffer);
  pushDrawCommand(commandBuffer,12);
}

// časovač
void Method::onUpdate(float dt){
  time += dt;
}

void Method::onDraw(SceneParam const&sceneParam){
  // výpočet matic
  auto lightView =
      glm::lookAt(glm::vec3(100*glm::cos(time),100,100*glm::sin(time)),glm::vec3(0,0,0),glm::vec3(0,1,0));
  auto lightProj = glm::ortho(-100.f,+100.f,-100.f,+100.f,0.f,1000.f);
  auto lightBias =
      glm::scale(glm::mat4(1.f),glm::vec3(.5f,.5f,1.f))*glm::translate(glm::mat4(1.f),glm::vec3(1,1,0));

  // nastavení uniformních proměnných
  mem.uniforms[0].m4 = sceneParam.view;
  mem.uniforms[1].m4 = sceneParam.proj;
  mem.uniforms[2].m4 = lightView;
  mem.uniforms[3].m4 = lightProj;
  mem.uniforms[4].m4 = lightBias;
  mem.uniforms[5].i1 = -1;
  mem.uniforms[7].v3 = glm::vec3(0.2f);
  mem.uniforms[8].v3 = glm::vec3(1.f);
  gpuRun(mem,commandBuffer);
}
```

Není třeba jej opisovat, vše je v podstatě uděláno, jen shadery můžete použít jako inspiraci.

15.4 Úkol 11 - finální render

15.4.1 Test 62 - finální render

```
izgProject -c --test 62
```

A je to! Gratuluji k vypracování celého projektu. Děkuji, že jste jej vypracovali celý. Teď by vám mělo fungovat vše.

Chapter 16

16 Rozdělení souborů a složek

Projekt je rozdělen do několika podsložek:

build/ Tady se čeká, že si budete sestavovat projekt, ale není to nutné, pokud víte, co děláte...

doc/ Tato složka obsahuje doxygen dokumentaci projektu. Můžete ji přegenerovat pomocí příkazu doxygen spuštěného v root adresáři projektu.

docSrc/ Tato složka obsahuje zdrojové kódy k dokumentaci.

examples/ Tato složka obsahuje přiložené příklady, které využívají vámi vytvořené zobrazovadlo.

framework/ Tato složka obsahuje interní záležitosti projektu. Všechny soubory jsou napsány v C++, abyste se mohli podívat, jak to funguje.

libs/ Tato složka obsahuje pomocné knihovny

resources/ Tato složka obsahuje modely a obrázky.

solutionInterface Tato složka obsahuje rozhraní/interface řešení. Slouží jako interface pro studentské řešení a učitelské řešení.

src/ Tato složka obsahuje main

studentSolution/ Tato složka obsahuje soubory, které budete editovat při implementaci projektu. Složka obsahuje soubory, které budete odevzávat a podpůrné hlavičkové soubory (které nebudete odevzdávat.

teacherSolutionBinary Tato složka obsahuje před kompilované učitelské řešení pro provádění testů.

tests/ Tato složka obsahuje akceptační a performanční testy projektu. Akceptační testy jsou napsány s využitím knihovny catch. Testy jsou rozděleny do testovacích případů (TEST_CASE). Daný TEST_CASE testuje jednu podčást projektu.

Složka studentSolution/src/studentSolution/ obsahuje soubory, které se vás přímo týkají:

studentSolution/src/studentSolution/gpu.cpp obsahuje definici funkce představující funkcionalitu grafické karty student_GPU_run - tady odvedete nejvíce práce.

studentSolution/src/studentSolution/prepareModel.cpp obsahuje definici funkce pro zpracování modelu student_prepareModel a vertex a fragment shaderu student_drawModel_vertexShader student_drawModel_fragmentShader - toto máte taky naprogramovat.

Složka solutionInterface/src/solutionInterface/ obsahuje podpůrné hlavičkové soubory, které se vás týkají:

solutionInterface/src/solutionInterface/gpu.hpp obsahuje definice typů a konstant pro grafickou kartu - projděte si. solutionInterface/src/solutionInterface/modelFwd.hpp obsahuje definice typů a konstant pro model - projděte si.

Chapter 17

17 Sestavení

Projekt byl testován na Ubuntu 20.04 - 24.04, Visual Studio 2017, 2019. Projekt vyžaduje 64 bitové sestavení. Projekt využívá build systém CMAKE. CMake je program, který na základně konfiguračních souborů "CMake↩ Lists.txt" vytvoří "makefile" v daném vývojovém prostředí. Dokáže generovat makefile pro Linux, mingw, solution file pro Microsoft Visual Studio, a další.
Postup Linux:

```
# stáhnout projekt
unzip izgProject.zip -d izgProject
cd izgProject/build
cmake ..
make -j8
./izgProject
./izgProject -h
```

Posup na Windows:

1. stáhnout projekt
2. rozbalit projekt
3. jděte do složky build/
4. ve složce build pusťte cmake-gui ..
5. pokud nevíte jak, tak pusťte cmake-gui a nastavte "Where is the source code:" na složku s projektem (obsahuje CMakeLists.txt)
6. a "Where to build the binaries: " na složku build
7. configure
8. generate
9. Otevřete vygenerovnou Microsoft Visual Studio Solution soubor.

Chapter 18

18 Spouštění

Projekt je možné po úspěšném přeložení pustit přes aplikaci **izgProject**. Projekt akceptuje několik argumentů příkazové řádky, pro jejich výpis použijte parametr **-h**

- **-c** spustí akceptační testy.
- **-p** spustí performanční test. (vhodné až pokud aplikaci zkompilujete v RELEASE) Vyzkoušejte si `./izgProject -h`

Chapter 19

19 Ovládání

Aplikace se ovládá pomocí myši a klávesnice:

- stisknuté levé tlačítko myši + pohyb myší - rotace kamery
- stisknuté pravé tlačítko myši + pohyb myší - přiblížení kamery
- stiknuté prostření tlačítko myši + pohyb myší - posun kamery do boků
- "n" - přepne na další scénu/metodu
- "p" - přepne na předcházející scénu/metodu
- "esc" - konec
- "wsadqe" - ovládání kamery
- "F9" - přepnutí na studentské řešení
- "F10" - přepnutí na učitelské řešení

Chapter 20

20 Testování

Vaši implementaci si můžete ověřit sadou vestavěných akceptačních testů. Když aplikaci pustíte s parametrem "-c", pustí se akceptační testy, které ověřují funkčnost vaší implementace.

```
./izgProject -c
```

Pokud není nějaký test splněn, vypíše se k němu komentář s informacemi, co je špatně. Testy jsou seřazeny a měly by se plnit postupně. Pokud chcete pustit jeden konkrétní test (třeba 13.), pusťte aplikaci s parametry "-c --test 13".

```
./izgProject -c --test 13
```

Pokud chcete pustit všechny testy až po jeden konkrétní (třeba 5.), pusťte aplikaci s parametry "-c --up-to-test --test 5".

```
./izgProject -c --test 5 --up-to-test
```

To je užitečné, když implementujete sekci, a chcete vědět, jestli jste něco zpětně nerozbili.
Na konci výpisu testů se vám vypíše bodové hodnocení.

Testování probíhá proti učitelskému řešení, které je přiloženo jako binárka v souborech:

- teacherSolutionBinary/windows/bin/libteacherSolution.dll
- teacherSolutionBinary/linux/lib/libteacherSolution.so

Je to pouze pro operační systémy Linux a Windows. MAC není podporován, protože nevlastním žádné takové zařízení a cross compilace na MAC by projekt ještě protáhla... Kdyby se vám tato dynamická knihovna nepodařila načíst, pak může být problém s verzí systému, v takovém případě se obraťte na Tomáše Mileta. Projekt byl testován na Merlinovi, kde to fungovalo.

Chapter 21

21 Odevzdávání

Odevzdejte **proj.zip**, který obsahuje jen soubory studentSolution/src/studentSolution/gpu.cpp a studentSolution/src/studentSolution/pr
žádné složky.
Můžete odevzdat částečné řešení, hodnotí se to, co jste odevzdali a kolik bodů vám to vypočetlo.

Před odevzdáváním si zkontrolujte, že váš projekt lze přeložit na merlinovi.
Pro ověření kompilace nemusíte na merlin kopírovat složku resources (je velká).
Pokud si chcete na merlinovi ověřit i akceptační testy stačí zkopírovat jen resources/models/parrots.glb.
Zkopirujte projekt na merlin a spusťte skript: **./merlinCompilationTest.sh**.

Odevzdávejte pouze soubory gpu.cpp, prepareModel.cpp Soubory zabalte do archivu proj.zip. Po rozbalení archivu se **NESMÍ** vytvořit žádná složka. Příkazy pro ověření na Linuxu:

```
zip -j proj.zip studentSolution/src/studentSolution/gpu.cpp
      studentSolution/src/studentSolution/prepareModel.cpp
```

Rozbalení:

```
unzip proj.zip
```

Studenti pracují na řešení projektu samostatně a každý odevzdá své vlastnířešení. Poraďte si, ale řešení vypracujte samostatně! Žádné kopírování kódu! Nedávejte svůj projekt veřejně na github, gitlab, soureforge, pastebin, discord nebo jinam, tím se automaticky stávate plagiátory.
Neposílejte svým kamarádům kódy. Možná jim věříte, že to neokopírují, ale divili byste se. Pak byste se dostali mezi plagiátory.

Chapter 22

22 Časté chyby, které nedělejte

1. student se mě nezeptá pokud neví, jak něco vyřešit. Ptejte se. Odpovím, pokud budu vědět.
2. student neodevzdá korektně zabalené soubory.
3. student si inkluduje nějake soubory z windows, třeba windows.h - to nedělejte, překlad musí fungovat na merlinovi.
4. student si přibalí nějaké náhodné soubory s MAC - to nedělejte, překlad musí fungovat na merlinovi.
5. min, max funkce si berete odnikud - vyzkoušejte, jestli vám jde překlad na merlinovi, nebo použijte glm::min, glm::max
6. špatně pojmenovaný archiv při odevzdávání
7. soubory navíc, nebo přejmenované soubory v odevzdaném archivu
8. memory corruption, přistupujete do paměti, kam nemáte (na to je valgrind)
9. student odevzdá soubory v nějakém exotickém archivu, rar, tar.gz, 7z, iso...
10. student zkouší projekt na systemu, který nebyl ověřen (ověřeno to bylo na Linuxu, Windows by měl běžet, ale ...).
11. VirtualBox s Ubuntu je +- možný, ale může se narazit na SDL chybu no video device (asi je potřeba nainstalovat SDL: sudo apt install xorg-dev libx11-dev libgl1-mesa-glx).
12. Nějaký problém se CMake a zprovoznením překladu na Windows (většinou je problém s cestami, zkuste dát projekt někam do jednoduché složky C:).
13. Projekt máte příliš pomalý a tak jej automatické testy předčasně utnou.

Chapter 23

23 Hodnocení

Množství bodů, které dostanete, je odvozeno od množství splněných akceptačních testů a podle toho, zda vám to kreslí správně (s jistou tolerancí kvůli nepřesnosti floatové aritmetiky). Automatické opravování má k dispozici větší množství akceptačních testů (kdyby někoho napadlo je obejít). Pokud vám aplikace spadne v rámci testů, dostanete 0 bodů. Pokud aplikace nepůjde přeložit, dostanete 0 bodů.

Chapter 24

24 Soutěž

Pokud váš projekt obdrží plný počet bodů, bude zařazen do soutěže o nejrychlejší implementaci zobrazovacího řetězce. Můžete přeimplementovat cokoliv v odevzdávaných souborech pokud to projde akceptačními testy a kompilací.
Spuštění měření výkonnosti:

```
./izgProject -p -f 10
```

Nejrychlejší projekty budou na věčné časy zařazeny do síně slávy. A...
Ceny za 1., 2. a třetí místo v roce 2023-2024 byly:

Cena za nejrychlejší projekt v roce 2024-2025 bude:
????

Chapter 25

25 Závěrem

Ať se dílo daří a ať vás grafika alespoň trochu baví! Omlouvám se, že zveřejnění projektu trvalo tak dlouho. Měl jsem na tom strašně moc práce a většinu nocí letního semestru kvůli tomu nespal. Šel jsem cestou kvalita, místo kvantita. V případě potřeby se nebojte zeptat (napište přímo vedoucímu projektu imilet@fit.vutbr.cz nebojte se napsat, nekoušu.