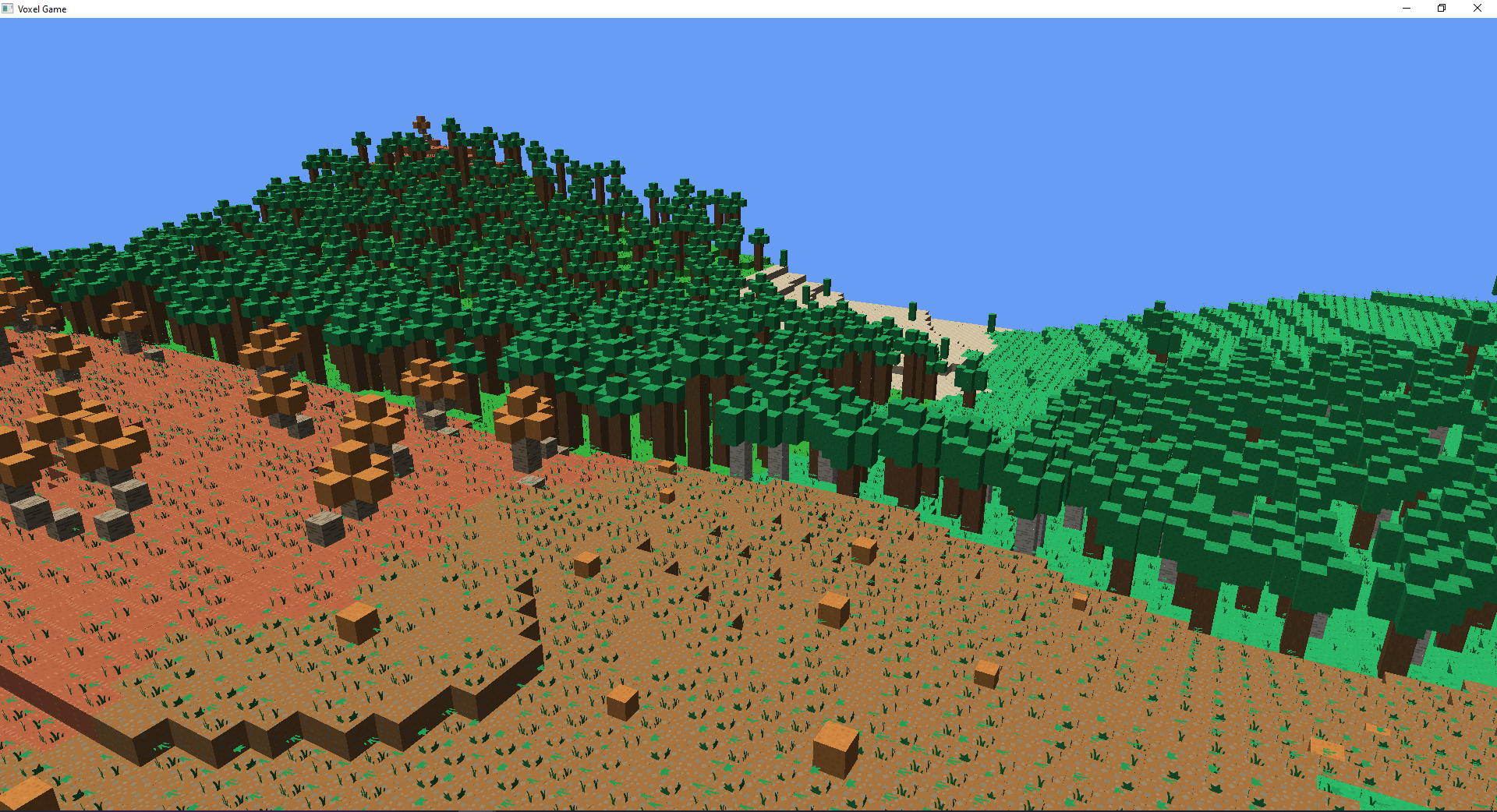
# Engine pro renderování a procedurální generování voxelových světů

## Modelování rostlin s využitím L-systémů

Herní svět obsahuje velké množství vegetace a ač jsou si všechny stromy, keře typově podobné, hráč si velice rychle všimne, že jsou identické. Na obrázku [obrázek Identické stromy] vidíme tři druhy stromů, které jsou zkopírované po scéně. Stromy v levé zadní části mají různou výšku, přesto působí umělým dojmem.

Stromy v reálném světě jsou si podobné -- rozeznáváme jednotlivé druhy stromů, přesto neexistují dva stejné stromy. Pokud druh stromu zapíšeme formální gramatikou nazývanou L-systém, docílíme podobné struktury stromů, které se budou lišit v detailech.

Identické stromy



<http://pcgbook.com/wp-content/uploads/chapter05.pdf>

### L-systém

L-systém nebo také Lindenmayerův systém je paralelní přepisovací systém vyvinutý maďarským teoretickým biologem a botanistou Aristidem Lindenmayerem v roce 1968. L-systém je typ formální gramatiky skládající se z abecedy, přepisovacích pravidel a počátečního axiomu. Pomocí postupného derivování počátečního axiomu je možné simulovat vývoj rostliny v čase.

### Interpretace řetězců pomocí želvy

Řetězce lze graficky reprezentovat pomocí želvy, konsumující symboly abecedy. Každý symbol určuje akci, kterou má želva vykonat. Želva se může pohybovat ve 2D nebo 3D prostoru. Ve 2D si můžeme interpretaci představit jako želvu, držící tužku, pohybující se po papíře.

[1.3 Turtle interpretation of strings http://algorithmicbotany.org/papers/abop/abop-ch1.pdf]

Želvu lze reprezentovat jako trojici (x, y, α), kde (x, y) představuje kartézské souřadnice reprezentující polohu v prostoru a α úhel kam želva směřuje. Zadáním délky kroku *d* a změny úhlu δ lze želvu ovládat pomocí následujících symbolů.

* F – Posun dopředu o délku *d*. Stav želvy se změní na (x’, y‘, α), kde x = x + d cos α a y = y + d sin α. Mezi body (x, y) a (x’, y‘) je nakreslena čára.
* + – Rotace doleva o úhel δ. Nový stav želvy (x, y, α + δ).
* - – Rotace doprava o úhel δ. Nový stav želvy (x, y, α - δ)

Nechť je definován následující L-systém. Buď ω počáteční axiom, *p* přepisovací pravidlo, δ = 90° a *d* zmenšené čtyřnásobně pro každý obrázek.

ω: F-F-F-F

p: F > F-F+F+FF-F-F+F

Želva interpretující daný L-systém generuje kvadratické Kochovy ostrovy. Obrázky jsou vygenerovány derivacemi o délce 0 až 3.

Diagram

Description automatically generated with medium confidence

[https://www.researchgate.net/figure/Quadratic-Koch-Island-by-L-System-a-n0-b-n1-c-n2-d-n3-Example-1-Let-us\_fig2\_258652653]

### Větvení v L-systémech

S danými přepisovacími pravidly není možné generovat větvící se struktury. Želva vždy pokračuje od své poslední pozice. Říše rostlin je dominovaná větvícími se strukturami, potřebujeme proto matematické vyjádření této skutečnosti.

(<http://algorithmicbotany.org/papers/abop/abop-ch1.pdf> 1.6 Branching structures)

Větvení v řetězci můžeme reprezentovat pomocí dvou symbolů [ a ], kde [ značí začátek větve a ] konec větve.

Symboly jsou interpretovány želvou následovně:

* [ -- Ulož atributy želvy do zásobníku.
* ] – Načti atributy želvy ze zásobníku a smaž je z vrcholu zásobníku (operace *pop*). Při této operaci není nakreslená žádná čára.

Díky nově přidaným symbolům lze generovat struktury připomínající rostliny. Struktury (obrázek) jsou generované následujícími L-systémy:

1. δ = 20°  
   ω: E  
   *p1*: F > FF  
   *p2*: E > F[+E]F[-E]+E
2. δ = 25,7°  
   ω: E  
   *p1*: F > FF  
   *p2*: E > F[+E][-E]FE

L-systém 1 generuje rostlinu vlevo, L-systém 2 generuje rostlinu vpravo.

Struktury připomínajících rostliny generované pomocí závorkovaného systémuA picture containing plant

Description automatically generated

### Stochastické L-systémy

(http://algorithmicbotany.org/papers/abop/abop-ch1.pdf 1.7 Stochastic L-systems)

Rostliny generované deterministickým L-systémem jsou všechny stejné. Jejich použití v scéně by vytvářelo stejný efekt, který je popsán na začátku kapitoly.

K předejití tohoto efektu je nutné zavést variace v rámci druhu. Náhodná interpretace řetězce má limitované využití. Změna úhlu větvení, šířky a výšky segmentů rostliny zachovávají topologii struktury, ze které je generovaná. Stochastické L-systémy mohou měnit topologii struktury.

L-systém, který byl do teď používán nemohl mít více přepisovacích pravidel pro stejný symbol abecedy. Pokud má stochastický L-systém vice přepisovacích pravidel, je z nich vybrána jedna s pravděpodobností 1/n, kde n je počet přepisovacích pravidel pro daný symbol abecedy[[1]](#footnote-1).

Scéna byla vygenerována za pomocí stochastického L-systému, kde:

δ = 25,7°  
ω: F  
*p1*: F > F[+F]F[-F]F  
*p2*: F > F[+F]F  
*p3*: F > F[-F]F

Využití stochastického L-systému.

A picture containing plant

Description automatically generated

### Implementace

#### Formát L-systému

L-systém může být načten ze souboru pomocí třídy *LSystemParser*. L-systém musí mít následující formát:

yaw\_angle pitch\_angle shring\_ratioaxiomletter > production...

letter > production

Soubor může obsahovat za sebou jdoucí L-systémy. *LSystemParser* je vrátí jako pole. Soubor může obsahovat komentáře na nových řádcích, začínající symbolem #.

Třída *LSystem* obsahuje gramatiku a tři atributy specifikující úhel náklonu podle osy y (yaw), podle osy x (pitch) a změnu velikosti bloku. Poslední atribut je využit při rozvětvení rostliny. Potomci mateřské větve by se měly řídit postulátem Leonarda da Vinci: „všechny větve stromu, v každé úrovni jeho růstu jsou v součtu jejich tloušťky rovné tloušťce kmene pod nimi.“ V případě dvojitého rozvětvení, tloušťky mateřské větve *w1*, tloušťky potomků *w2* dostaneme rovnici:

[Zdroj: <http://algorithmicbotany.org/papers/abop/abop-ch2.pdf> Modeling of trees, strana 57]

Hodnotu 0,7 je možné nalézt v L-systémech modelující keře.

#### Želva

Třída *Turtle* rozšiřuje pohyb želvy – popsané v kapitole Interpretace řetězců pomocí želvy – do 3D prostoru. Vnitřní stav želvy určují následující atributy:

* Pozice v prostoru.
* Velikost bloku vytvořeného želvou.
* Barva použitá pro kreslení (výstupní pole).
* Yaw – rotace podle osy y.
* Pitch – rotace podle osy x.

**TODO: tato část je stejná pro kameru**

(Zdroj: https://learnopengl.com/Getting-started/Camera)

Želva si udržuje tři navzájem kolmé směrové vektory (nahoru, dopředu, doprava) jednotkové délky, které využívá pro pohyb po scéně. Vektory jsou aktualizované po každé rotaci. Pro výpočet je nutné znát vektor směrující kolmo vzhůru vůči scéně (WORLD\_UP). Generovaný svět je plochý, proto lze tento vektor nahradit konstantním vektorem (0, 1, 0).

glm::vec3 front;  
front.x = cos(glm::radians(Yaw\_)) \* cos(glm::radians(Pitch\_));  
front.y = sin(glm::radians(Pitch\_));  
front.z = sin(glm::radians(Yaw\_)) \* cos(glm::radians(Pitch\_));  
Front\_ = glm::normalize(front);

Right\_ = glm::normalize(glm::cross(Front\_, WORLD\_UP));  
Up\_ = glm::normalize(glm::cross(Right\_, Front\_));

Délku *x* v rovině určené osami *x* a *z* lze spočítat jako délku přilehlé odvěsny. Cos(Yaw) = x / h, kde h je délka přepony. Víme, že vektor má jednotkovou délku, proto h = 1. Stejný postup aplikujeme pro rovinu určenou osami *x* a *y*.

Tímto způsobem dopočítáme délky *y* a *z* vektoru směrujícího dopředu a normalizujeme ho. Jelikož jsou na sebe vektory kolmé, využijeme vektorového součinu, jehož výsledkem je vektor kolmý k oběma původním vektorům. Všechny vektory je nutné normalizovat, aby se předešlo jejich zkracování s tím, jak se Pitch blíží +- 90°.

K zamezení převrácení os jsou z definičního oboru Pitch vyjmuty násobky 90°.

if (Helpers::Math::Equal(cos(glm::radians(Pitch\_)), 0.0f))  
 Pitch\_ -= 0.01f;

Výsledná nepřesnost je menší než maximální rozdíl dvou čísle typu float ε, která jsou považována za stejná. Funkce *Equal* porovnává desetinná čísla s přesností ε.

Želva vystavuje metody pro pohyb ve všech třech osách využívající vektorů *Up\_, Right\_, Forward\_*. K pozici želvy je přičten patřičný vektor naškálovaný délkou pohybu. Např.:

void LSystems::Detail::Turtle::MoveForward(float dz) {  
 Position\_ += Front\_ \* dz;  
}

#### Rozšířená abeceda

Následující symboly abecedy mají speciální význam pro jejich interpretaci.

* U a u – Posuň želvu nahoru.
* F a f – Posuň želvu dopředu.
* x – Zmenši želvu.
* X – Zvětši želvu.
* S – Nastav původní velikost želvy.
* + – Rotuj želvu doleva podle osy y.
* - – Rotuj želvu doprava podle osy y.
* ^ – Rotuj želvu nahoru podle osy x.
* & – Rotuj želvu dolů podle osy x.
* [ – Ulož kopii želvy na vrchol zásobníku.
* ] – Vyjmi želvu z vrcholu zásobníku.
* 0-9 – Přepni výstupní pole.

L-systém může obsahovat jakýkoliv jiný ASCII symbol – mimo bílých znaků a # – určený pro expanzi přepisovacích pravidel. Není želvou interpretován.

#### Ovládání želvy

Implementace želvy se nachází ve jmenném prostoru *LSystems::Detail*, uživatel by ji neměl využívat přímo, ale je pro něj připravena třída *LSystemExecutor* zajišťující generování herních objektů z poskytnutého L-systému.

*LSystemExecutor* umožňuje generovat herní objekty na základě stochastického L-systému – topologie struktury výsledného modelu se může měnit mezi jednotlivými voláními generátoru na základě parametru *salt*. L-systém lze náhodně interpretovat na základě těchto parametrů:

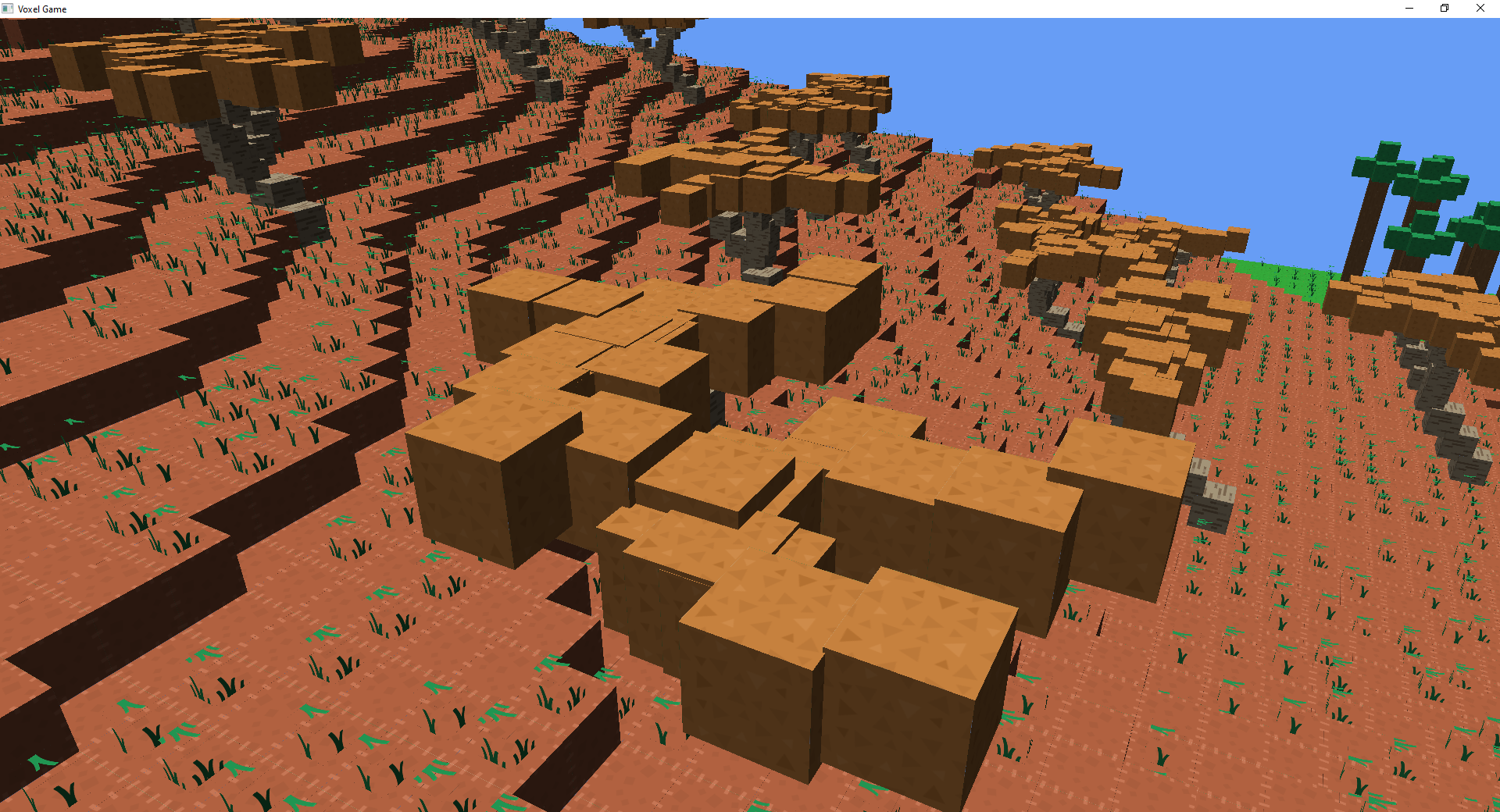
* Rozsah počtu provedených derivací.
* Variace v rotaci želvy. K úhlu, o který se má želva otočit, se přičte x \* původní úhel, kde x je z [-angleVar, angleVar]. Defaultní hodnota angleVar je 0,2.
* Výchozí velikost generovaných objektů. Lze určit rozsahem korespondujícím s počtem derivací.

Přidání náhodného úhlu má výrazný efekt na organický vzhled rostliny. Na obrázku níže lze vidět akáciové stromy vyznačující se plochou korunou. V definici L-systému jsou všechny listy ve stejné výšce, výsledný rozdíl ve výškách je způsoben opakovaným rotováním želvy. V zápisu lze vidět, že se želva otočí o 45° nahoru (^), pokládá větve (u, U), skloní se o 45° (&) a pokládá listy (F). Listy by tak měly být ve stejné rovině, ale nejsou.

L-systém generující akácie:

45.0 45.0 0.8  
# make sure the plant has splits  
# random lenght stem - then split  
^uA1S&F+F+F+F  
U > uU  
A > +uuE  
A > -uE  
A > +uE  
A > -E  
A > uuE  
A > uE  
# top of the plant  
E > x[++++UE1S&F+F+F+F]+UE  
E > x[++++++UE1S&F+F+F+F]++UE  
E > x[++UE1S&F+F+F+F]-UE

Nepravidelná koruna akáciového stromu



Výstupem generátoru je 2D pole obsahující herní objekty rozdělené podle čísla výstupního bufferu, který měla želva při generování. Část enginu je tak odstíněna od textur, které jsou definované v části procedurálního generátoru. Díky tomuto rozdělení modelu je možné snadno měnit textury pro jednotlivá pole. Procedurální generátor tohoto využívá a používá stejný model – jiný běh generátoru – pro vytváření bříz a dubů, lišících se texturou kmene a listů.

### Modelování rostlin

Při modelování vegetace bylo třeba velkého množství pokusů a ladění, kdy rostlina nevypadala přirozeně, ale nebylo jasné, v jaké časti gramatiky je problém. Nejvíce se mi osvědčila technika nalezení reálné rostliny obrázek xx a následné pokusy o její napodobení obrázek yy.

Velice se osvědčilo pravidlo:

U > uU

Díky němuž jsou větve blíže k zemi delší než větve navazující na korunu stromu. Pokud má rostlina význačné části je vhodné je modelovat samostatně (kmen, větve, koruna) viz [L-systém generující akácie].

Akáciový strom rostoucí v přírodě



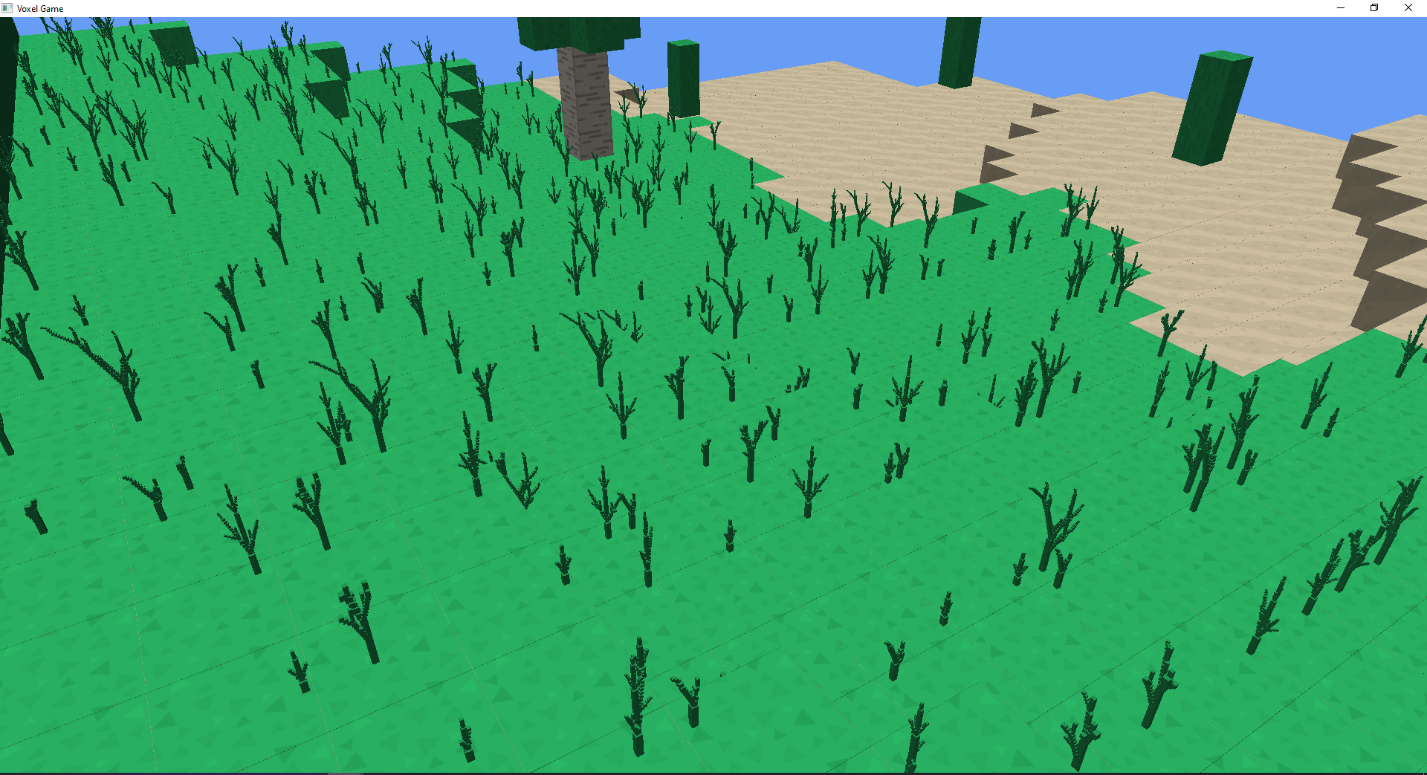
[Zdroj: <https://commons.wikimedia.org/wiki/File:Umbrella_thorn_acacia_or_israeli_babool_tree_plant_acacia_tortillis.jpg>]

Model akáciového stromu



Tato technika ne vždy přinášela ovoce. Modelování trávy rostoucí na planinách se projevilo jako problém. Tráva neměla dostatečnou hustotu a nezapadala do kresleného vzhledu obrázek xx. S navyšujícím se počtem herních objektů dramaticky rostla spotřeba paměti. Jeden herní objekt s texturou trávy byl nahrazen desítkami herních objektů, z kterých se skládal model trávy. Tento problém by mohl být řešen přesunem vytváření modelu na grafickou kartu, přidáním vertexů v geometry shaderu.

Tráva na planinách



Výsledné modely svou topologií připomínaly strukturu keřů. Byly proto upraveny – přidáním listí, změnou větvení – a využity v generátoru keřů.

#### Simulace růstu

Křovinatý biotop je porostlý dvěma druhy keřů rozděleným do třech fází růstu. Ty jsou simulovány opakovaným derivováním počátečního axiomu. Keře nejmenšího vzrůstu jsou derivovány 2x, největší keře jsou derivovány 4x. Tloušťka kmene koreluje lineárně s počtem provedených derivací. Větší stromy mají širší kmen a dorůstají vyšší výšky.

Každý druh si zachovává své typické vlastnosti. Na obrázku[[2]](#footnote-2) lze pozorovat stejné zakončení větví – rozdělení do dvou větví rostoucích na opačné strany – a podobný úhel v jakém se větve oddělují od kmene. Topologie rostliny se díky stochastickému L-systému mění mezi jednotlivými jedinci.

Díky těmto krokům biotop obsahuje rostliny navzájem podobného vzhledu, lišících se v drobných detailech.

Fáze růstu keře



### Výsledky použití L-systémů

Definice L-systémů jsou krátké (~ 10 řádků na jeden druh rostliny) a produkují velké množství rozdílných jedinců. Předchozí manuální definování rostlin v kódu bylo náročnější, a i při přidání více jedinců pro každý druh, by bylo snadné najít stejné. Odebráním definic rostlin z kódu se zvýšila jeho čitelnost – definice L-systémů jsou zdroje dat, který engine konzumuje. Nutnost kompilace při změně modelu byla odstraněna a zvýšila se rychlost iterace, s kterou je možné upravovat model.

Vytvořením vlastního formátu pro zápis modelu rostliny se oddělila závislost na programovacím jazyce. Modely tak může vytvářet jiný člen týmu bez znalosti programování a překladu kódu.

Generováním rostlin za běhu programu se snížila rychlost jeho běhu. Tento problém lze mitigovat cachováním rostlin obsahujících velké množství herních objektů. Toto bylo provedeno pro keře. Před spuštěním generace terénu je naplněn buffer obsahující keře vygenerované na základě seedu. Buffer musí být dostatečně velký na to, aby nedošlo ke snížení diverzity rostlin. Při vytváření keře na scéně je vybrán náhodný index do bufferu, závislý na pozici keře. Vybraný model je zkopírován a přesunut na dané místo.

Při porovnání scény ze začátku kapitoly si lze všimnout přirozenějšího vzhledu krajiny. Koruny stromů se mohou překrývat, scéna díky tomu působí více organicky – stromy v přírodě nemají přesně stanové hranice, kde končí jeden a začíná druhý. Výsledná scenérie působí méně jednolitě díky rozdílným vývojovým stádiím rostlin.

Vegetace vygenerovaná na základě L-systémů



## Herní engine

**TODO: historie herních enginů?**

Herní engine zajišťuje vykreslování scény a komunikaci s grafickou kartou, zpracování vstupu od uživatele, správu zdrojů. Poskytuje třídy s obecnou funkcionalitou, využívané generátorem terénu. Mezi ně patří:

* Práce s náhodnými jevy.
* Načítání a generování L-systémů.
* Továrna pro vytváření herních objektů.
* Správa textur.

### Herní smyčka

Moderní grafické programy nezpracovávají data dávkově, ale obvykle čekají na vstup od uživatele, který následně zpracují. Na rozdíl od většiny softwaru, hry běží i když uživatel neposkytuje žádný vstup. Hra nezamrzne, animace se vykreslují, monstra se pohybují po scéně. Klíčovou částí je neblokující zpracování vstupu. (zdroj http://gameprogrammingpatterns.com/game-loop.html)

while (true) {

processInput();

update();

render();

}

Základními kameny jsou zpracování vstupu, co se stal od posledního volání *processInput. update* posune herní simulaci o jeden krok. *render* na závěr vše vykreslí na obrazovku. Zde vyvstává otázka, jak rychle herní smyčka běží.

#### Variabilní časový krok

Mějme dva hráče. První má výkonný počítač a hra běží rychlostí 60 FPS (snímků za vteřinu). Druhý hráč má méně výkonný počítač a hra běží rychlostí 6 FPS. Avatar druhého hráče by se pohyboval pouze desetinovou rychlostí. Rychlost hry je přímo závislá na rychlosti hardwaru.

Pokud nemáme kontrolu nad hardwarem, na kterém hra běží, je toto řešení nepřípustné. Implementace musí brát v potaz, jak dlouho trvá jedna iterace herní smyčky, a podle ní zvětšit nebo zmenšit krok o který bude simulace posunuta vpřed.

while (!game->Finished()) {

const auto currentFrame = static\_cast<float>(glfwGetTime());

deltaTime = currentFrame - lastFrame;

lastFrame = currentFrame;

game->ProcessInput(deltaTime);

game->Update(deltaTime);

game->Render();

}

Metodám měnícím herní stav je předána hodnota *deltaTime*. Vykreslení zachycuje stav scény v okamžiku zavolání, proto nezávisí na *deltaTime*.

### Vykreslování scény

**TODO: world coords, etc.**

Vykreslovací engine má za úkol převod scény definované v 3D prostoru, na 2D plochu zobrazovacího zařízení hráče. Tento proces začíná na CPU, kde jsou herní objekty převedeny na data (matice), která jsou poslána grafické kartě. Zde jsou za pomocí série kroků transformována na zobrazitelné pixely. Tyto kroky jsou vysoce specializované a výstup každého z nich je použit jako vstup pro další.

Tyto kroky je možné masivně paralelizovat a využít tisíců jader, které může grafická karta obsahovat. Každé jádro spouští malý program, pro každý krok vykreslovacího řetězce. Tyto programy jsou nazývané shadery. Některé ze shaderů může definovat vývojář a nahradit nimi existující výchozí shadery. OpenGL pro jejich programování využívá OpenGL shading language (GLSL).

(Zdroj: <https://learnopengl.com/Getting-started/Hello-Triangle>)

#### Předání dat grafické kartě

Scéna je složena z kusů terénu (třída *Chunk*), obsahující herní objekty. Chunky jsou rozmístěné na ploše vymezené osami x a *z*. Třída *Scene* má za úkol správu chunků a serializaci jejich dat do 1D pole, předané grafické kartě.

Každý herní objekt lze reprezentovat jako matici 4x4 obsahující informace o posunu vůči počátku světa, škálování a textuře.

**TODO: model matrix**

Převod herního objektu na matici:

for (const auto& o : obs) {

assert(o.HasComponent<Components::Transform>());

auto model = o.GetComponent<Components::Transform>().ModelMat();

assert(o.HasComponent<Components::SpritesheetTex>());

const auto& texPos = o.GetComponent<Components::SpritesheetTex>().GetTexPos();

Helpers::Math::PackVecToMatrix(model, texPos);

buffer[cube]->push\_back(model);

}

[[nodiscard]] glm::mat4 ModelMat() const {

auto model = glm::mat4(1.0f); // identity matrix

model = glm::translate(model, Position);

return glm::scale(model, Scale);

}

##### Instancing

Herní scéna obsahuje velké množství objektů se stejnou geometrií (např. krychle), s různou transformací vůči počátku světa. Každá krychle je složena z 12 trojúhelníků. Její vykreslení je téměř okamžité. Pokud by pro každou z nich bylo voláno samostatné vykreslení (draw call), velice rychle doje k drastickému snížení výkonu (komunikace s kartou přes sběrnici, uložení dat do patřičných bufferů, …). Lepší řešení je poslat všechna data kartě najednou a ty následně vykreslit pomocí stejné geometrie. Toto řešení se nazývá instancing.

(Zdroj: https://learnopengl.com/Advanced-OpenGL/Instancing)

Scéna umožňuje definovat vlastní geometrie – krychle, krychle bez podstav, nebo jakákoliv jiná kombinace stěn krychle. Pro každou z nich alokuje místo na grafické kartě, uloží do něj data objektů a následně samostatně vykreslí každou z nich.

// bind data

glBindBuffer(GL\_ARRAY\_BUFFER, InstanceDataBufferIds\_[cube]);

unsigned offset = 0;

for (const auto& chunk : instancesData) {

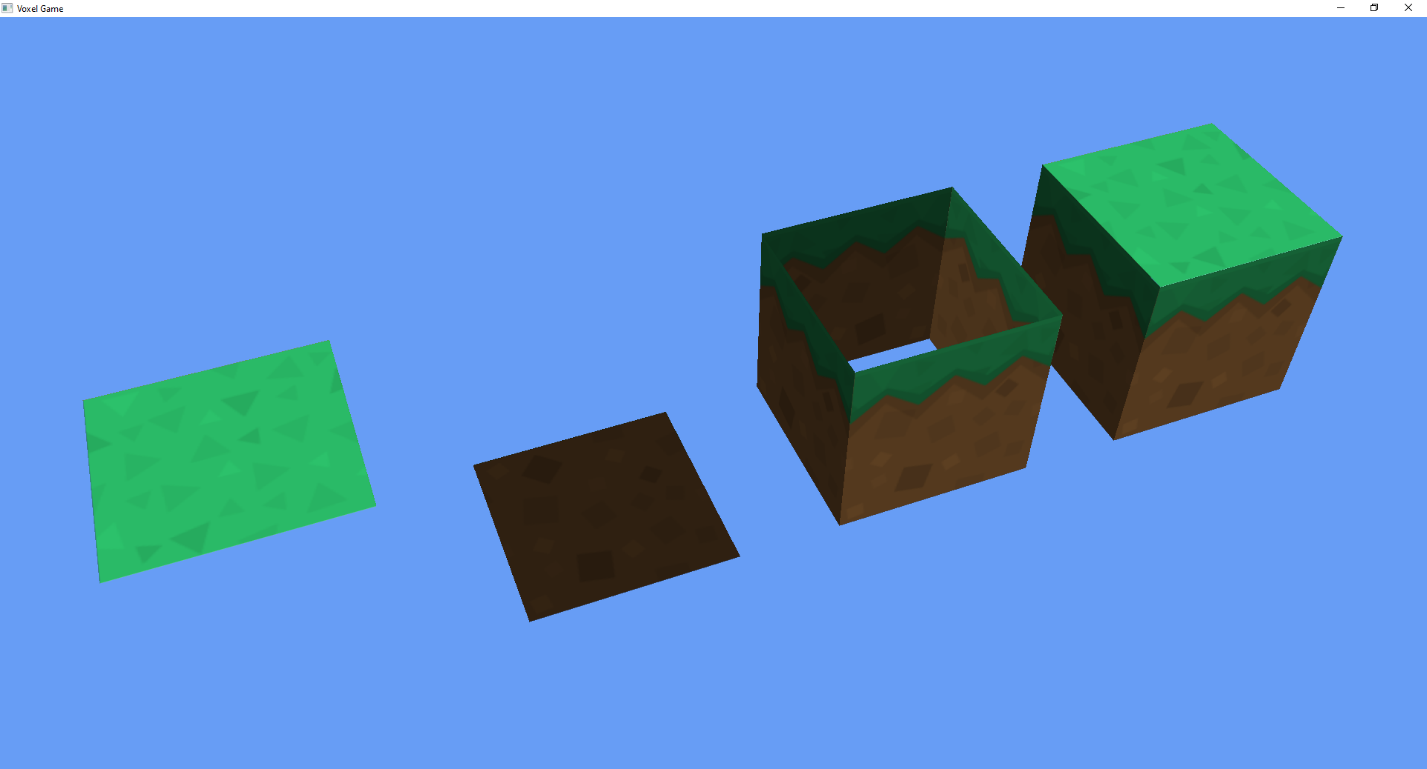
glBufferSubData(GL\_ARRAY\_BUFFER, offset \* sizeof(glm::mat4), chunk->size() \* sizeof(glm::mat4), chunk->data());

offset += chunk->size();

}

CubeRenderers\_[cube].GetDefaultMesh().BindBatchAttribPtrs();

Rozdílné geometrie jsou užitečné, pokud chceme definovat krychli s rozdílnými podstavami a stěnami. Na obrázku xx, lze vidět blok trávy složený ze tří částí.



#### Společná data objektů

Instancing umožňuje grafické kartě používat společná data pro všechny vykreslované objekty. Jeho protějškem v objektovém světe je návrhový vzor Flyweight.

Pokud má být objekt vykreslitelný musí obsahovat komponentu *Transform* (určující jeho pozici a velikost)a komponentu *Mesh* (představující geometrii objektu) – obrázek xx.

**Obrázek herních komponent**

*Mesh* obsahuje informace o vrcholech (*Vertex)* vykreslovaných trojúhelníků – pozici na scéně, normálový vektor k úsečce mezi dvěma vrcholy trojúhelníku, pozici na textuře. A pole indexů, které říká, z jakých vrcholů jsou trojúhelníky složeny. Každý vrchol v má velikost 8 \* 4 = 32 B a *Mesh* jich obsahuje 6 \* 4. Každá stěna krychle má čtyři různé vrcholy[[3]](#footnote-3). Pole indexů má velikost 12 \* 3 \* 4 (počet trojúhelníků, počet vrcholů trojúhelníku, velikost *unsigned*). Celková velikost *Mesh*e představující krychli je:

8 \* 4 \* 6 \* 4 + 12 \* 3 \* 4 = 912 B

Pokud má být schopný engine vykreslovat desetitisíce objektů je tato paměťová náročnost neúnosná. Komponentu *Mesh* není možné odebrat z herního objektu, protože obsahuje pozici na textuře, lišící se mezi objekty. Pozice na textuře představuje čtverec, který bude vybrán z textury a aplikován na povrch krychle. Všechny objekty mají texturu čtverce. Díky tomu je možné definovat čtverec na počátku textury a jeho posun uložit do samostatné komponenty -- *SpritesheetTex*. Toto nové rozložení je znázorněno na obrázku xx.

**Obrázek nových herních komponent**

Komponenta *SpritesheetTex* obsahuje dvě čísla ve formátu float (float je zvolen pro jednodušší komunikaci s grafickou kartou – všechna předaná data mají formát float). Komponenta v paměti zabírá pouhých 8 B. Velikost herního byla redukována o 904 B.

*Mesh* není dále používána jako komponenta herního objektu. Její implementace ze jmenného prostoru *Renderer*, kterou komponenta *Mesh* obsahuje jako svůj atribut, je použita pro vykreslení všech objektů se stejnou geometrií najednou. Vykreslení objektů zajišťuje třída *CubeRenderer*, viz obrázek xx.

**TODO: zapínání a vypínaní face culling. Použití u trávy, glitch u celého bloku při Z-fighting**

**TODO: blending, prolínání alfa kanálu s pozadím**

**TODO: shaders**

**TODO: světlo**

### Zpracování vstupu

O vstup z klávesnice a myši se stará abstraktní třída *InputHandler*, obsahující pole příkazů (třída *Command*) ovládajících kameru. Konkrétní implementaci má na starosti třída *InputHandlerGl*, pevně svázaná s voláními OpenGL.

Vstup je zpracován dvojím způsobem:

1. *ProcessInput*, voláno v rámci herní smyčky.
2. Zpětné volání z OpenGL.

Pohyb hráče zajišťuje metoda *ProcessInput*. Délka pohybu je škálována parametrem *delta*. Vstupy jsou zpracovány při volání metody, kontrolou kláves, které jsou právě stlačené. Tento způsob je ideální pro kontinuální změnu stavu (např. pozice) objektu. Hráč může klávesu držet pro plynulý pohyb, nebo ji opakovaně mačkat pro drobné korekce.

Tento způsob je naprosto nevhodný pro změnu stavu nabývajícího několika diskrétních hodnot. Například vypnutí/zapnutí světla. Hráč musí klávesu stisknout a pustit po dobu trvání jednoho snímku – při 60~FPS na 16,67~ms. Při běžném stisknutí může dojít k několikanásobné změně stavu.

#### Zpětné volání z OpenGL

K zamezení opakovaného čtení stisknuté klávesy lze využít nastavení funkce, kterou OpenGL zavolá při stisknutí klávesy. Signatura této funkce musí být:

void function\_name(GLFWwindow\* window, int key, int scancode, int action, int mods)

(zdroj: <https://www.glfw.org/docs/3.3/group__input.html#ga1caf18159767e761185e49a3be019f8d>)

Výhody třídní hierarchie (dědění z obecné třídy pro zpracování vstupu a její konkrétní implementace pro OpenGL, možnost mít více objektů kontrolujících vstup a jejich výměna pro změnu ovládaní) znemožňují nastavení přímého volání metody z OpenGL. Zpětné volání zajišťují funkce umístěné ve jmenném prostoru *Input::Detail*.

namespace Input::Detail {  
void CursorPosCallback(GLFWwindow\*, double xPos, double yPos) {  
 currentHandler->ProcessMouse(static\_cast<float>(xPos), static\_cast<float>(yPos));  
}

void ScrollCallback(GLFWwindow\*, double, double yOffset) {  
 currentHandler->ProcessMouseScroll(static\_cast<float>(yOffset));  
}

void KeyCallback(GLFWwindow\*, int key, int scanCode, int action, int mods) {  
 currentHandler->ProcessKey(key, scanCode, action, mods);  
}  
} // namespace Input::Detail

Tyto funkce nejsou umístěné v hlavičkovém souboru a uživatel by je neměl napřímo využívat. Zpětné volání je automaticky nastaveno při inicializaci objektu.

void Input::InputHandlerGl::SetCallBacks() {  
 Detail::currentHandler = this;  
 WindowManagerGl::SetCursorPosCallback(Detail::CursorPosCallback);  
 WindowManagerGl::SetScrollCallback(Detail::ScrollCallback);  
 WindowManagerGl::SetKeyCallback(Detail::KeyCallback);  
}

Jmenný prostor *Input::Detail* obsahuje pouze jednu proměnnou, označující současný objekt zpracovávající vstup.

#### Návrhový vzor příkaz

Jednoduchá implementace kódu zpracovávajícího vstup by mohla vypadat:

**TODO: obrázek klávesnice s voláním metod**

void InputHandler::handleInput() {

if (isPressed(BUTTON\_W)) object.MoveForward();

else if (isPressed(BUTTON\_S)) object.MoveBackward();

else if (isPressed(BUTTON\_A)) object.MoveLeft();

else if (isPressed(BUTTON\_D)) object.MoveRight();

}

Tento kód funguje, pokud jsme ochotni natvrdo zadrátovat vstup hráče k herním akcím viz obrázek xx, ale hodně her hráče nechá nakonfigurovat si vlastní mapování tlačítek. K tomuto potřebujeme vyměnit přímé volání metody *A* za metodu *B*. K tomu potřebujeme objekt reprezentující volání metody.

(zdroj: <http://gameprogrammingpatterns.com/command.html>)

Definujme abstraktní třídu *Command* představující spustitelný příkaz.

class Command {

protected:

Renderer::Camera\* Actor;

public:

explicit Command(Renderer::Camera\* actor) : Actor(actor) {}

virtual ~Command() = default;

virtual void Execute(float delta) = 0;

};

V konkrétní implementaci příkazu je z něj poděděno a metoda *Execute* přepsána pro specifické chování.

class MoveForwardCommand : public Command {

public:

explicit MoveForwardCommand(Renderer::Camera\* actor) : Command(actor) {}

void Execute(float delta) override {

Actor->Move(delta, 0.0, 1.0);

}

};

*InputHandler* je odstíněn od volání prováděných na *actor* (obrázek xx), sníží se tak spojení mezi třídami a zvýší se soudržnost (high cohesion) *InputHandler –* má na starosti zpracování vstupu, ne ovládání herní postavy.

**TODO: Přidat diagram tříd. High cohesion**

Příkazy jsou uloženy v poli std::array<std::unique\_ptr<Commands::Command>, Keys::Count> Commands. Toto pole je naplněno při inicializaci *InputHandleru,* neobsahuje tedy žádné *nullptr* ukazatele. Při odstranění příkazu reagujícího na stlačení tlačítka, by při jeho stisku došlo k vyhození *nullptr* výjimky a pádu programu. Toto chování může být ošetřeno důslednou kontrolou obsahu *unique\_ptr*, byl by tím ovšem porušen objektový návrh. Řešení použité v enginu je *NullCommand*.

class NullCommand : public Command {

public:

NullCommand() : NullCommand(nullptr) {}

explicit NullCommand(Renderer::Camera\*) : Command(nullptr) {}

void Execute(float) override {}

};

Toto řešení zachovává principy objektového návrhu. Přijímá parametr *actor*, svému předku však předá *nullptr*. Tělo metody *Execute* je prázdné a s *actorem* není interagováno. Volání metody *Execute* mohou být bezpečně provedena na všech prvcích pole.

## Zkratky

FPS – frames per second

1. Tato definice se liší, od definice uvedené ve (zdroj). Tento způsob náhodného výběru je použit v implementaci, kde pravděpodobností distribuci zastupuje několikanásobné zopakování přepisovacího pravidla. [↑](#footnote-ref-1)
2. Pro větší názornost byly odstraněny části modelu představující listy. [↑](#footnote-ref-2)
3. Krychle má 8 vrcholů. Vrcholy jednotlivých stěn se ale liší v normálovém vektoru a pozicí na textuře. [↑](#footnote-ref-3)