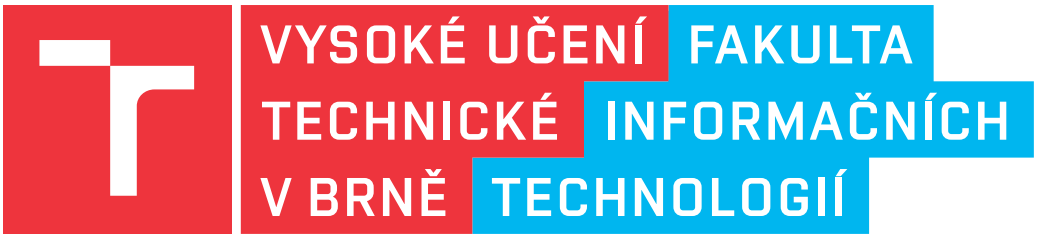


Projekt do předmětu GMU – Grafické a multimediální procesory

Akcelerace procedurálního generování pomocí GPU

15. prosince 2018

Řešitelé: Daniel Čejchan (xcejch00@stud.fit.vutbr.cz)
Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

1 Zadání

Cílem této práce je implementace vybrané metody procedurálního generování jako algoritmus pro GPU. Rychlost algoritmu bude srovnána s alternativní implementací. Výsledek procedurálního generování má být adekvátně vizualizován.

2 Použité technologie

Implementace byla realizovaná pomocí *compute shaderu* v OpenGL 4.3. Byla vytvořena demonstrační aplikace v jazyce D[1] s využitím knihoven BindBC-opengl[2] (rozhraní pro OpenGL) a DSFML [3] (správa okna, uživatelských vstupů a vykreslování 2D primitiv a textu). Způsob sestavení aplikace je popsán v souboru `README.md` v kořenovém adresáři projektu.

3 Použité zdroje

Pro základní studium algoritmů byla využita wikipedie. Dodatečné znalosti byly získány z článků Stefana Gustavsona[4] a Kena Perlina[5]. Stefan Gustavson také pod MIT licencí na githubu uveřejnil svou implementaci Perlinova a Simplex šumu v OpenGL *shaderu*[6]. Tato implementace byla zavedena do demonstrační aplikace pro účely srovnání výkonu.

4 Způsob akcelerace

Jako algoritmus pro implementaci a optimalizaci byl zvolen Perlinův šum. Akceleraci realizujeme implementací metody jakožto *compute shader*. *Shader* bude zapisovat do 3D textury; každému voxelu v textuře přiřadí hodnotu šumu v rozsahu $[0, 255]$. Kód výsledného *shaderu* lze najít v souboru `res/shader/optimizedPerlin.cs.glsl` a část v `res/shader/inc/common.glsl`.

4.1 Perlinův šum

Výpočet hodnoty Perlinova šumu probíhá následovně: nad prostorem je zavedena jednotková ortogonální mřížka. Pro uzly mřížky je pseudonáhodně určena hodnota $\overrightarrow{grad}$ (D je dimenze šumu):

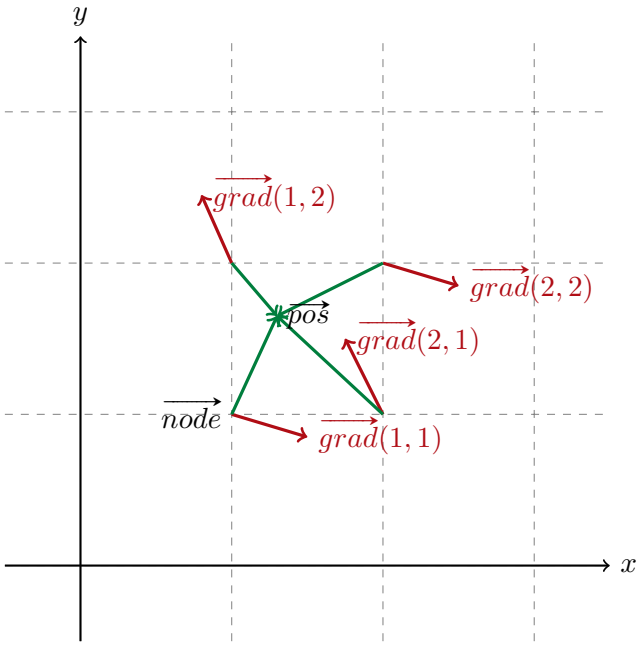
$$\overrightarrow{grad}(\overrightarrow{pos}, seed) : \mathcal{Z}^D \times \mathcal{N} \to \mathcal{Z}^D. \tag{1}$$

Pro 2D šum jsou doporučeny vektory jednotkové délky, pro 3D Perlin určuje 12 vektorů (jedná se o vektory do středů hran jednotkové krychle), ze kterých se má pseudonáhodně vybírat.

[1] http://dlang.org
[2] https://github.com/BindBC/bindbc-opengl
[3] http://dsfml.com/
[4] Gustavson, S.: Simplex noise demystified. 2005.
[5] Perlin, K.: Improving Noise. ACM Transactions on Graphics (TOG), ročník 21, č. 3, 2002: s. 681–682, ISSN 1557-7368.
[6] https://github.com/ashima/webgl-noise

Z mřížky se kolem bodu $\overrightarrow{pos}$, pro který chceme zjistit hodnotu šumu, vybere 2^D uzlů $\overrightarrow{node}_{i\in[1,2^D]}$ tvořící vrcholy D-rozměrné krychle s délkou strany jedna. Pro každý z těchto uzlů je vypočtena hodnota dot jako

$$dot_i(\overrightarrow{pos}, seed) = (\overrightarrow{pos} - \overrightarrow{node_i}) \cdot \overrightarrow{grad}(\overrightarrow{node_i}, seed). \tag{2}$$



(3) Vizualizace gradientů a mřížky u 2D Perlinova šumu

Výsledná hodnota šumu je potom interpolací mezi hodnotami $dot_{i\in 2^D}$:

$$perlin(\overrightarrow{pos}, seed) = \sum_{i=1}^{2^D} dot_i(\overrightarrow{pos}) * \prod_{j=1}^{D} ipol(1 - |\overrightarrow{pos}[j] - \overrightarrow{node_i}[j]|). \tag{4}$$

Protože pracujeme s jednotkovou mřížkou, $|\overrightarrow{pos}[j] - \overrightarrow{node_i}[j]|$ bude vždy ležet v intervalu $[0, 1]$. Jako interpolační funkci $ipol$ Perlin definuje

$$ipol(x \in [0, 1]) = 6x^5 - 15x^4 + 10x^3, \tag{5}$$

která má pro body $x = 0$ a $x = 1$ nulovou první a druhou derivaci.

Pro esteticky zajímavější výsledky se šum běžně kombinuje v různých měřítkách a amplitudách:

$$moPerlin(\overrightarrow{pos}, seed) = \sum_{o=1}^{O} perlinNoise(\overrightarrow{pos} * scaling^o, seed) * persistence^{(o-1)}, \tag{6}$$

kde O je počet kombinovaných šumů, $scaling$ je úbytek měřítka (např. 0.5) a $persistence$ je úbytek amplitudy (např. 0.8).

Tento výpočet je spuštěn pro každý bod, pro který chceme zjistit hodnotu šumu. Pro 128^3 voxelů to tedy znamená 128^3 výpočtů na scénu. Výpočty lze provádět zcela paralelně.

Při implementaci se zaměříme na 3D Perlinův šum. Velikost pracovní skupiny v *compute shaderu* určíme fixně na $8 * 8 * 8$ tak, aby všechny tři dimenze měly stejný, co největší rozměr (velikost pracovní skupiny 16^3 by přesahovala `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`). Každá invokace bude vypočítávat hodnotu šumu pro jeden voxel ve scéně.

Dříve v tomto oddíle byla mřížka u Perlinova šumu definována jako jednotková. Dále budeme zdánlivě pracovat s různými velikostmi mřížky. Toho je docíleno tak, že skutečné souřadnice bodu ve scéně vždy vydělíme velikostí mřížky $\overrightarrow{pos} = \overrightarrow{realPos}/gridSize$ – vnitřní implementace tedy vždy pracuje s jednotkovou mřížkou. Velikost mřížky budeme nazývat velikost oktávy.

4.2 Optimalizace 1: Výpočet gradientů

Pro lepší optimalizovatelnost přijmeme omezení, že velikost oktávy je vždy dělitelná rozměrem pracovní skupiny (tedy osmi), a je tedy současně i větší než je rozměr pracovní skupiny. V důsledku toho všechny body v pracovní skupině vždy spadají do stejné krychle určené mřížkou. To znamená, že gradienty (ve 3D je jich 8, dle počtu vrcholů krychle) budou mezi všemi invokacemi v pracovní skupině stejné, a dají se tedy předpočítat.

Pro pseudonáhodné generování definujme následující hashovací funkci (převzato z internetu):

```
1  uint hash(uint x) {
2    x = ((x >> 16) ^ x) * 0x45d9f3b;
3    x = ((x >> 16) ^ x) * 0x45d9f3b;
4    x = (x >> 16) ^ x;
5    return x;
6  }
```

Pro výpočet gradientu definujeme funkci `nodeGradient`. Ta má pro každý uzel mřížky pseudonáhodně vybrat jeden z těchto dvanácti gradientů:

$$(1,1,0), (-1,1,0), (1,-1,0), (-1,-1,0),$$
$$(1,0,1), (-1,0,1), (1,0,-1), (-1,0,-1),$$
$$(0,1,1), (0,-1,1), (0,1,-1), (0,-1,-1)$$

Pro zarovnání na počet 16 Perlin doporučuje (konkrétní) čtyři z těchto gradientů dát do výběru dvakrát. Základní možná implementace je tato:

```
1  const int gradientCount = 16;
2
3  const vec3 gradients[gradientCount] = {
4    vec3(1,1,0), vec3(-1,1,0),
5    ...
6  };
7
8  vec3 nodeGradient(ivec3 pos) {
9    uint t = hash(uint(pos.x));
10   t = hash(t ^ uint(pos.y));
11   t = hash(t ^ uint(pos.z));
12
13   return gradients[t % gradientCount];
14 }
```

Všimněme si ale, že ve vektoru jsou vždy některé dvě souřadnice buď 1 nebo -1 a třetí je nulová. Můžeme tedy zkusit odstranit čtení z paměti. Řádek 13 nahradíme za:

```
1  vec3 result = vec3(
2  1 - float((t << 1) & 2),
3  1 - float(t & 2),
4  1 - float((t >> 1) & 2)
5  );
6  result[(t >> 3) % 3] = 0; // One dimension is always 0
7  return result;
```

Zde nám maskování `(t & 2)` navrátí buď dvojku, nebo nulu. Tedy `1 - (t & 2)` bude buď 0, nebo -1. Takto pseudonáhodně vypočteme hodnoty všech tří složek gradientu a nakonec určíme jednu složku, kterou nastavíme na nulu. Výsledek tedy bude ekvivalentní s naší předchozí implementací. Bylo experimentálně ověřeno, že tato varianta je mírně rychlejší.

Další možnou oblastí pro optimalizaci je využití kooperace vláken při výpočtu gradientů. Bez kooperace vláken by si každá invokace musela hodnoty všech gradientů vypočítat, tedy by kód vypadal nějak takto:

```
1  const float gradient1 = nodeGradient(anchorNodePos);
2  const float gradient2 = nodeGradient(anchorNodePos + vec3(1,0,0));
3  const float gradient3 = nodeGradient(anchorNodePos + vec3(0,1,0));
4  const float gradient4 = nodeGradient(anchorNodePos + vec3(1,1,0));
5  const float gradient5 = nodeGradient(anchorNodePos + vec3(0,0,1));
6  const float gradient6 = nodeGradient(anchorNodePos + vec3(1,0,1));
7  const float gradient7 = nodeGradient(anchorNodePos + vec3(0,1,1));
8  const float gradient8 = nodeGradient(anchorNodePos + vec3(1,1,1));
```

Proměnná `anchorNodePos` zde reprezentuje ten z vrcholů mřížkové krychle, který je nejblíž počátku souřadného systému (vypočte se jako `floor(pos)`). V tomto případě by ve všech invokacích pracovní skupiny běžel paralelně vždy stejný výpočet. Deklarujeme tedy sdílenou proměnnou `cachedGradient` a kód upravíme tak, aby v jednom kroku každá invokace vypočetla hodnotu jiného gradientu:

```
1  shared vec3 cachedGradients[OCTAVE_COUNT * 8];
2
3  if(gl_LocalInvocationIndex < 8 * OCTAVE_COUNT) {
4    // Smallest octave size is 8, doubles each octave
5    const int octaveSize = 8 << (gl_LocalInvocationIndex >> 3);
6    const ivec3 octaveAnchorPos = pos / octaveSize;
7
8    const ivec3 offset = octaveAnchorPos + ivec3(
9    (gl_LocalInvocationIndex & 1),
10   (gl_LocalInvocationIndex >> 1) & 1,
11   (gl_LocalInvocationIndex >> 2) & 1
12   );
13
14   cachedGradients[gl_LocalInvocationIndex] = nodeGradient(offset);
15 }
```

Ve výsledcích níže v tomto dokumentu uvidíme, že tato optimalizace výpočet značně urychlí. Protože máme $8^3 = 512$ invokací a potřebujeme vypočítat pouze 8 gradientů, velká část pracovní skupiny by zůstala nečinná. Proto můžeme v tomto kroku vypočítat gradienty až pro $512/8 = 64$ oktáv najednou.

4.3 Optimalizace 2: Výpočet dot_i

Dalším krokem v algoritmu je pro každý vrchol krychle výpočet hodnoty dot_i dle rovnice 2:

$$dot_i(\overrightarrow{pos}, seed) = (\overrightarrow{pos} - \overrightarrow{node_i}) \cdot \overrightarrow{grad}(\overrightarrow{node_i}, seed).$$

Hodnoty funkce $\overrightarrow{grad}(\overrightarrow{node_i}, seed)$ již máme předpočítané v $cachedGradients$ (v naší implementaci neuvažujeme $seed$). Výraz upravíme zadefinováním $\overrightarrow{offset^i} = \overrightarrow{pos} - \overrightarrow{node_i}$ a rozepsáním skalárního součinu:

$$\begin{aligned} dot_i(\overrightarrow{pos}) = &(\overrightarrow{pos}_x - \overrightarrow{node}^i_x) * \overrightarrow{cachedGradient}[i]_x \\ &+ (\overrightarrow{pos}_y - \overrightarrow{node}^i_y) * \overrightarrow{cachedGradient}[i]_y \\ &+ (\overrightarrow{pos}_z - \overrightarrow{node}^i_z) * \overrightarrow{cachedGradient}[i]_z. \end{aligned} \tag{7}$$

Všimněme si, že každý prvek součtu závisí vždy pouze na jedné složce vektoru pozice. Tudíž pro všechny invokace se stejnou pozicí v x, ale libovolnými pozicemi v y a z (tedy pro $8 * 8 = 64$ různých invokací) je $(\overrightarrow{pos}_x - \overrightarrow{node}^i_x) * cachedGradient^i_x$ stejné; analogicky toto platí i pro další dvě složky. Tyto prvky součtu můžeme tedy v rámci pracovní skupiny vypočítat jen jednou a uložit si je do proměnné:

$$cachedDotData[i][j] = ((j, j, j) - \overrightarrow{node_i}) * cachedGradient[i] \tag{8}$$

$$\begin{aligned} dot_i(\overrightarrow{pos}) = &\, cachedDotData[i][\overrightarrow{pos}_x]_x \\ &+ cachedDotData[i][\overrightarrow{pos}_y]_y \\ &+ cachedDotData[i][\overrightarrow{pos}_z]_z. \end{aligned} \tag{9}$$

Vhodnou implementací lze opět vypočítat tyto parametry najednou až pro 8 oktáv:

```
1  // 8 interpolations, OCTAVE_COUNT octaves, 8 local size
2  shared vec3 cachedDotData[8][OCTAVE_COUNT][8];
3
4  if(gl_LocalInvocationIndex < OCTAVE_COUNT * 64) {
5    const uint offsetId = (gl_LocalInvocationIndex & 7);
6    const uint gradientId = (gl_LocalInvocationIndex >> 3) & 7;
7    const uint octaveId = gl_LocalInvocationIndex >> 6;
8
9    const uint octaveSize = 8 << octaveId;
10   // pos - gl_LocalInvocationID = workgroup origin
11   const vec3 offset = fract(vec3(pos - gl_LocalInvocationID +
       offsetId) / octaveSize);
12
13   const vec3 gradient = cachedGradients[octaveId << 3 |
       gradientId];
14
15   cachedDotData[gradientId][octaveId][offsetId] = vec3(
16   (offset.x - (gradientId & 1)) * gradient.x,
17   (offset.y - ((gradientId >> 1) & 1)) * gradient.y,
18   (offset.z - ((gradientId >> 2) & 1)) * gradient.z
19   );
20 }
```

V kódu výše máme vektor `offset`, který udává rozdíl aktuální pozice a pozice $\overrightarrow{node_1}$, tedy vrcholu krychle, který je nejblíž k počátku souřadného systému. Všechny složky vektoru $offset$

leží v intervalu $[0, 1]$. Protože pozice $\overrightarrow{node_1}$ se vypočte jako `floor(pos)`, `offset` se dá vypočíst jako `fract(pos)` (v OpenGL je `fract(x)` ekvivalentní `x - floor(x)`).

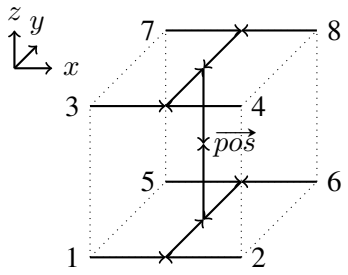
Výsledné skalární součiny se potom vypočítají jako:

```
float dotProduct[8];

for(int i = 0; i < 8; i++) {
  dotProduct[i] =
  cachedDotData[i][octaveId][gl_LocalInvocationID.x].x
  + cachedDotData[i][octaveId][gl_LocalInvocationID.y].y
  + cachedDotData[i][octaveId][gl_LocalInvocationID.z].z;
}
```

4.4 Interpolace

Posledním krokem je interpolace mezi jednotlivými $dotProduct[i \in [0, 8)]$. Kompaktnost zápisu v rovnici 4 se dá nahradit intuitivnějším výkladem: budeme postupně interpolovat mezi gradienty dvou vrcholů, kde poměr interpolace bude udávat pozice bodu $\overrightarrow{pos}$ vůči těmto dvěma vrcholům. Interpolujme nejprve dvojice vrcholů po ose x, poté interpolujeme mezivýsledky po ose y a nakonec podle z:



(10) Vizualizace interpolace u Perlinova Šumu

V GLSL existuje pro interpolaci funkce `mix`. Předáním vektoru můžeme provést interpolaci pro více hodnot najednou (toto nemá žádný měřitelný vliv na výkon, pouze to umožňuje kompaktnější zápis):

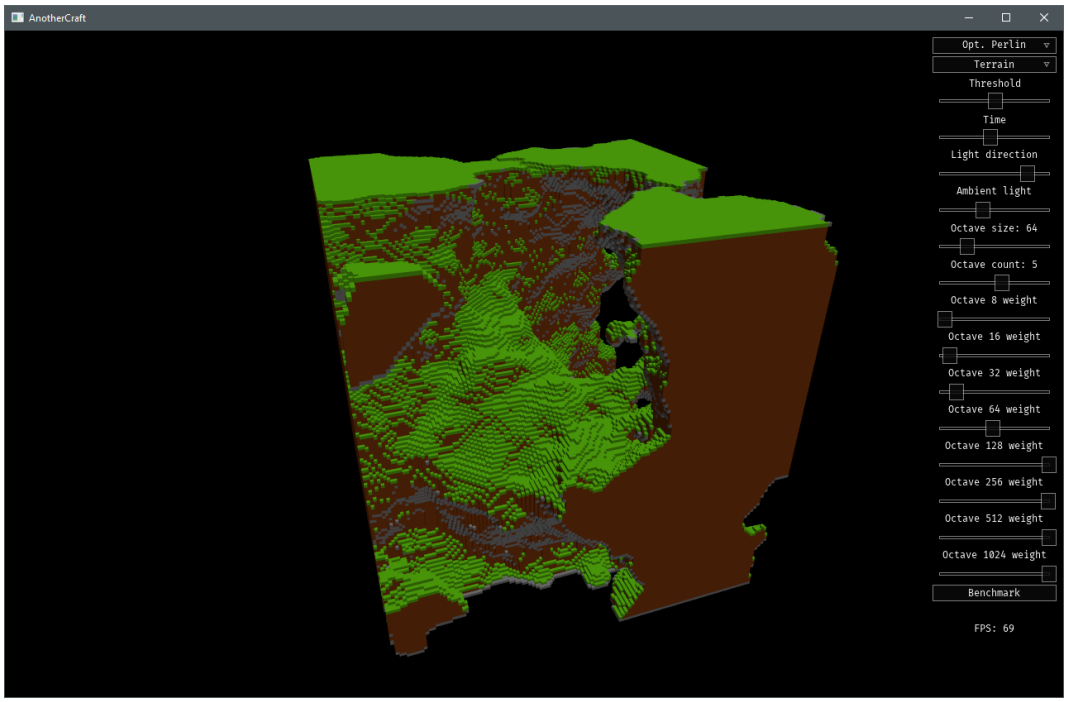
```
const vec4 interpolation1 = mix(
vec4(dotProduct1, dotProduct3, dotProduct5, dotProduct7),
vec4(dotProduct2, dotProduct4, dotProduct6, dotProduct8),
interpolationConst(offset.x)
);

const vec2 interpolation2 = mix(
vec2(interpolation1[0], interpolation1[2]),
vec2(interpolation1[1], interpolation1[3]),
interpolationConst(offset.y)
);

const float result = mix(
interpolation2.x,
interpolation2.y,
interpolationConst(offset.z)
);
```

5 Ovládání vytvořeného programu

Výsledky procedurálního generování byly vizualizovány formou voxelového terénu (scéna s 128^3 voxely), protože se to slučovalo s autorovými záměry pro diplomovou práci.



(11) Screenshot demonstrační aplikace

Aplikace zobrazuje generovanou voxelovou scénu, kterou lze tahem myší otáček a kolečkem myši přibližovat/oddalovat. GUI vpravo umožňuje volbu generujícího algoritmu, nastavení metody obarvování voxelů a nastavení parametrů pro generování (ne všechny parametry jsou aplikovány v každé metodě). Posuvník "Threshold"určuje prahovou hodnotu generující funkce, nad kterou jsou odpovídající voxely zobrazeny.

Generující algoritmy jsou následující:

1. **Opt. Perlin, Hopt. Perlin, Unopt. Perlin**: 3D Perlinův šum v různých úrovních optimalizace
2. **[Ashima], [Ashima simplex]**: Implementace Perlinova a Simplex šumu Stefanem Gustavstonem[7], přejato z githubu
3. **3D+T Opt. Perlin, 2D+T Opt. Perlin**: 2D/3D Perlinův šum s parametrem času
4. **2D Opt. Perlin**: 2D Perlinův šum
5. **1oct Perlin, 1oct Simplex**: Jednooktávový Perlinův a Simplex šum (lze nastavit pouze "Octave size", "Octave XX weight"nic nedělá)
6. **2D Voronoi, 3D Voronoi**: Metody založené na Voroného diagramech

[7] https://github.com/ashima/webgl-noise/

6 Vyhodnocení

Všechna měření byla prováděna na notebooku Acer Aspire V15 Nitro II (VN7-592G) s následujícími specifikacemi:

- Procesor Intel Core i7-6700HQ; 2,6 GHz, 4 jádra, 8 vláken
- RAM 8GB DDR4, 2133 MHz
- GPU NVIDIA GeForce GTX 960M; architektura Maxwell, 4 GB GDDR5

Měření výkonu bylo realizováno na CPU tímto způsobem:

```
/* Setup everything */
glFinish();
/* Start stopwatch here */
glDispatchCompute(...);
glFinish();
/* Read stopwatch here */
```

Na GPU byl šum vždy vypočtem dvěstěkrát, výpočet šumu následoval jeden zápis do 3D textury. Každému z dvou set výpočtů byly předloženy jiné parametry, aby se v co největší míře zabránilo optimalizacím napříč jednotlivými výpočty.

```
void main() {
  float val = 0;

  // Weirds constants there are to make sure that each execution
     is different
  for(int i = 0; i < params.executionCount; i++)
  val = val * 0.000001 +
    noiseFunction(ivec3(gl_GlobalInvocationID.xyz) + i * 1271);

  val = clamp(0.5 + val/2, 0, 1);
  imageStore(world, ivec3(gl_GlobalInvocationID.xyz),
    uvec4(uint(round(val * 255)), 0, 0, 0));
}
```

Měření probíhalo vždy pro 1–8 oktáv (makro `OCTAVE_COUNT`) pro tyto algoritmy:

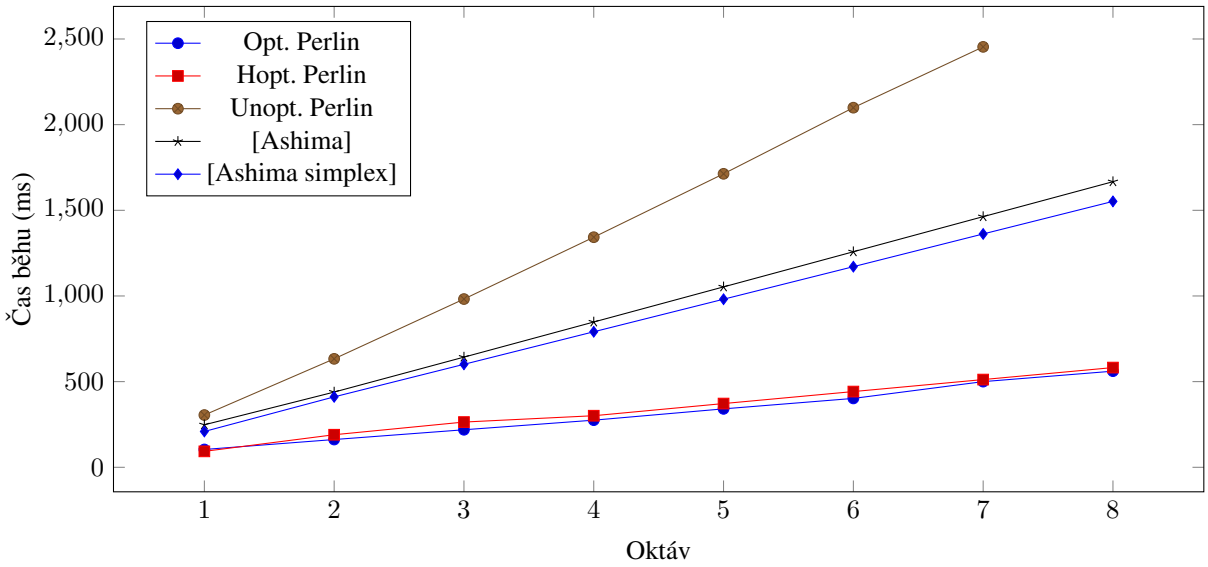
1. **Opt. Perlin**: Naše maximálně optimalizovaná verze
2. **Hopt. Perlin**: Naše verze bez optimalizace 2, stále využívající vláknovou kooperaci při výpočtu gradientů (optimalizaci 1)
3. **Unopt. Perlin**: Naše verze bez vláknové kooperace
4. **[Ashima]**: Gustavsonova verze, bez vláknové kooperace
5. **[Ashima simplex]**: Gustavsonova implementace *Simplex noise* (ten by měl být rychlejší jak Perlin), bez vláknové kooperace

Měření probíhalo na hodnotách implicitně nastavených při startu aplikace. Všechny výsledky uvádějí běh celého výpočtu (200* algoritmus + zápis do textury) v milisekundách. Měření se v aplikaci spouští stiskem tlačítka ”Benchmark”. Výsledky vykazovaly stabilitu $\pm$2ms při opakovaném měření.

Oktáv:	1	2	3	4	5	6	7	8
Opt. Perlin	**104**	**162**	**219**	**275**	**341**	**402**	**500**	**561**
Hopt. Perlin	93	190	264	301	372	442	512	582
Unopt. Perlin	305	633	982	1343	1713	2099	2454	–
[Ashima]	248	439	643	848	1053	1258	1463	1667
[Ashima simplex]	209	411	601	791	981	1171	1362	1552

(12) Tabulka doby běhů jednotlivých algoritmů (ms na 200 běhů).

(13) Graf doby výpočtu šumové funkce v závislosti na počtu oktáv



Výsledky ukazují, že využití vláknové kooperace v našem případě vykazuje více než třikrát větší rychlost výpočtu oproti stejnému algoritmu bez kooperace. Implementace Stefana Gustavsona je přibližně o 50 % rychlejší než naše verze bez vláknové kooperace, oproti naší verzi s vláknovou kooperací je však 3x pomalejší. Podstatné urychlení poskytla optimalizace 1 (kooperativní výpočet hodnot gradientů; Unopt. Perlin vs. Hopt. Perlin). Komplexnější optimalizace 2 (Opt. Perlin vs Hopt. Perlin) už na výkon příliš velký vliv neměla.

7 Doporučení pro budoucí zadávání projektů

Projekt byl zajímavý a bavil mě. Práce by mne ale těšila mnohem víc, kdybych nebyl také zaneprázdněn další prací do školy. Dle mého názoru rozsah projektu značně přesahuje dotaci jak bodovou, tak i kreditovou (kdyby nebyla zkouška, tak by se to ještě dalo pochopit). Jelikož se dala práce spojit se semestrálním projektem, ještě se to docela dalo zvládnout. Kdybych měl ale zadání diplomky neslučitelné (což jsem původně měl), tak upřímně nevím, jestli bych tento semestr zvládl.