Wprowadzenie do technologii HDR

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

Rendering HDR

Tradycyjnie w grach komputerowych do renderowania grafiki używamy 8 bitów na kanał. W ten sposób jesteśmy w stanie przechować 256 różnych wartości i uzyskać kontrast 256 : 1. Przez wiele lat było to wystarczającym zakresem.

Z czasem grafika ewaluowała i powstały coraz bardziej złożone i realistyczne algorytmy do generowania grafiki w grach. 256 wartości na kanał przestało dawać wystarczająco duży zakres, aby otrzymać grafikę zadowalającej jakości. Zatem zaczęto korzystać z możliwości jakie daje HDR.

Domyślny bufor ramki korzysta z bufora koloru, w którym przypada 8 bitów na kanał. Zatem żeby móc skorzystać z możliwości HDR musimy mieć możliwość renderowania do bufora z większą liczbą bitów na kanał. Do tego celu będziemy używać formatów zmiennoprzecinkowych, które nie tylko dadzą nam większy zakres wartości, ale również większą precyzję, co jest istotne m.in. przy obliczeniach oświetlenia czy dynamicznym kontraście.

Żeby zaimplementować HDR musimy wykonać następujące kroki:

- stworzyć zmiennoprzecinkowy bufor (tekstura, obiekt bufora obrazu itp.),
- wyrenderować scenę z wszystkimi efektami (oświetlenie, teksturowanie, efekty specjalne itp.) do bufora z poprzedniego punktu,
- wyrenderować efekty specjalne HDR,
- wyrenderować czworokąt pokrywający całe okno lub ekran nakładając na niego teksturę będącą zawartością użytego bufora zmiennoprzecinkowego, ponadto wykonujemy mapowanie tonów oraz korekcję gamma jeśli korzystamy z urządzenia LDR.

Obiekty bufora obrazu

Biblioteka OpenGL obok możliwości rysowania grafiki w oknie lub na całym ekranie oferuje rysowanie w tzw. obiektach bufora obrazu (ang. framebuffer object – FBO).

Domyślny obiekt FBO jest wiązany z tworzonym oknem. Możemy utworzyć wiele obiektów FBO i renderować obrazy w nich zamiast w oknie. Jest to tzw. metoda renderowania pozaekranowego (ang. off-screen rendering).

Technika ta stosowana jest do otrzymania wielu różnych efektów, np. mapowanie cieni, odbicie, przetwarzanie końcowe (ang. postprocessing).

Generowanie identyfikatora obiektu FBO:

```
void glGenFramebuffers(GLsizei n, GLuint *ids);
```

`n` – liczba identyfikatorów do wygenerowania, `ids` – tablica na wygenerowane identyfikatory.

Dowiązanie obiektu FBO:

```
void glBindFramebuffer(GLenum target, GLuint framebuffer);
```

`target` określa rodzaj obiektu FBO i przyjmuje wartości: `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, `GL_FRAMEBUFFER`, zaś `framebuffer` to identyfikator obiektu FBO.

W danej chwili możemy mieć dowiązany tylko jeden obiekt FBO do rysowania i jeden do odczytu.

Usuwanie obiektów FBO:

```
void glDeleteFramebuffer(GLsizei n, const GLuint *framebuffers);
```

`n` – liczba obiektów FBO do usunięcia, `framebuffers` – tablica z identyfikatorami obiektów FBO do usunięcia.

Po utworzeniu obiektu FBO musimy do niego dowiązać obiekty buforów renderowania (ang. renderbuffer object – RBO).

Generowanie identyfikatorów obiektów RBO:

```
void glGenRenderbuffers(GLsizei n, GLuint *renderbuffers);
```

`n` – liczba identyfikatorów do wygenerowania, `renderbuffers` – tablica na wygenerowane identyfikatory.

Usuwanie obiektów RBO:

```
void glDeleteRenderbuffers(GLsizei n, const GLuint *renderbuffers);
```

`n` – liczba obiektów RBO do usunięcia, `renderbuffers` – tablica z identyfikatorami obiektów RBO do usunięcia.

Dowiązanie obiektu RBO:

```
void glBindRenderbuffer(GLenum target, GLuint renderbuffer);
```

`target` przyjmuje wartość `GL_RENDERBUFFER`, a `renderbuffer` to identyfikator obiektu RBO.

Po dowiązaniu obiektu RBO musimy przydzielić mu pamięć:

```
void glRenderbufferStorage(GLenum target, GLenum internalformat, GLsizei
 width, GLsizei height);
```

`target` przyjmuje wartość `GL_RENDERBUFFER`, `width`, `height` to wymiary (w pikselach) obiektu RBO, a `internalformat` to format wewnętrzny: `GL_RED`, `GL_RG`, `GL_RGB`, `GL_RGBA`, `GL_RGB8`, `GL_RGBA8` itp., `GL_DEPTH_COMPONENT`, `GL_DEPTH_COMPONENT16`, `GL_DEPTH_COMPONENT24`, `GL_DEPTH_COMPONENT32`, `GL_DEPTH_COMPONENT32F`, `GL_DEPTH24_STENCIL8`, `GL_DEPTH32F_STENCIL8`, `GL_DEPTH_STENCIL`, `GL_STENCIL_INDEX`, `GL_STENCIL_INDEX1`, `GL_STENCIL_INDEX4`, `GL_STENCIL_INDEX8`, `GL_STENCIL_INDEX16` itp.
W przypadku HDR interesującymi dla nas wartościami `internalformat` będą m.in. `GL_RGB16F`, `GL_RGBA16F`, `GL_RGB32F`, `GL_RGBA32F`.

Dowiązywanie obiektu RBO do obiektu FBO:

```
void glFramebufferRenderbuffer(GLenum target, GLenum attachment, GLenum
renderbuffertarget, GLuint renderbuffer);
```

`target` przyjmuje wartość `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, `GL_FRAMEBUFFER` (to samo znaczenie co `GL_DRAW_FRAMEBUFFER`), `renderbuffertarget` przyjmuje wartość `GL_RENDERBUFFER`, `renderbuffer` to identyfikator obiektu RBO, zaś `attachment` to tzw. punkt wiązania: `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT`, `GL_DEPTH_STENCIL_ATTACHMENT`, `GL_COLOR_ATTACHMENT0`, `GL_COLOR_ATTACHMENT1`, `GL_COLOR_ATTACHMENT2`, ... Maksymalną liczbę `GL_COLOR_ATTACHMENT` możemy odczytać ze stałej `GL_MAX_COLOR_ATTACHMENTS`.

Przed użyciem obiektu FBO z dowiązanymi obiektami RBO musimy sprawdzić kompletność bufora obrazu:

```
GLenum glCheckFramebufferStatus(GLenum target);
```

`target` przyjmuje jedną z wartości: `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, `GL_FRAMEBUFFER` (oznacza to samo co `GL_DRAW_FRAMEBUFFER`). Jeśli funkcja zwróci `GL_FRAMEBUFFER_COMPLETE`, to wszystko jest w porządku. W przypadku gdy coś jest nie tak funkcja zwróci jedną z wartości:

- `GL_FRAMEBUFFER_UNDEFINED` – bieżące dowiązanie FBO ma numer 0, ale nie istnieje bufor domyślny,
- `GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT` – jeden z buforów przeznaczonych do renderowania jest niekompletny,

- `GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT` – brak buforów powiązanych z FBO,
- `GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER` – z jednym z dowiązanych buforów do renderowania nie związano żadnego bufora,
- `GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER` – z jednym z dowiązanych buforów do odczytu nie związano żadnego bufora,
- `GL_FRAMEBUFFER_UNSUPPORTED` – nieobsługiwana kombinacja formatów wewnętrznych,
- `GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` – liczba próbek wszystkich buforów nie jest zgodna,
- `GL_FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS` – niektóre dowiązania kolorów nie są warstwowymi teksturami.

Renderowanie do tekstury

Do obiektu FBO oprócz obiektów RBO możemy dowiązać teksturę:

```
void glFramebufferTexture1D(GLenum target, GLenum attachment, GLenum
textarget, GLuint texture, GLint level);
```

```
void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum
textarget, GLuint texture, GLint level);
```

```
void glFramebufferTexture3D(GLenum target, GLenum attachment, GLenum
textarget, GLuint texture, GLint level, GLint layer);
```

`target` przyjmuje jedną z wartości: `GL_DRAW_FRAMEBUFFER`, `GL_READ_FRAMEBUFFER`, `GL_FRAMEBUFFER` (oznacza to samo co `GL_DRAW_FRAMEBUFFER`), `attachment` to punkt wiązania, `textarget` określa typ tekstury (np. `GL_TEXTURE_2D`), `texture` to identyfikator obiektu tekstury, `level` to poziom mipmapy, `layer` to warstwa tekstury 3D.

Rendering do wielu buforów

Aby możliwe było wysyłanie danych kolorów do wielu buforów musimy w shaderze fragmentów generować odpowiednią liczbę wartości.

Pierwszym sposobem na zwrócenie tych danych jest zapisanie ich we wbudowanej zmiennej wyjściowej o nazwie `gl_FragData`. Jest to tablica typu `vec4`.

Drugim sposobem jest zdefiniowanie kilku zmiennych wyjściowych w shaderze fragmentów. Musimy również określić lokalizacje tych zmiennych. W tym celu przed linkowaniem programu cieniowania używamy funkcji:

```
void glBindFragDataLocation(GLuint program, GLuint colorNumber, const
char *name);
```

`program` – identyfikator programu cieniowania, `colorNumber` – lokalizacja, `name` – nazwa zmiennej wyjściowej.

Zamiast funkcji `glBindFragDataLocation` możemy skorzystać z kwalifikatora formatu `layout(location = n)`, który umieszczamy przy zmiennej wyjściowej shadera fragmentów.

Drugim sposobem jest zdefiniowanie kilku zmiennych wyjściowych w shaderze fragmentów. Musimy również określić lokalizacje tych zmiennych. W tym celu przed linkowaniem programu cieniowania używamy funkcji:

```
void glBindFragDataLocation(GLuint program, GLuint colorNumber, const
char *name);
```

`program` – identyfikator programu cieniowania, `colorNumber` – lokalizacja, `name` – nazwa zmiennej wyjściowej.

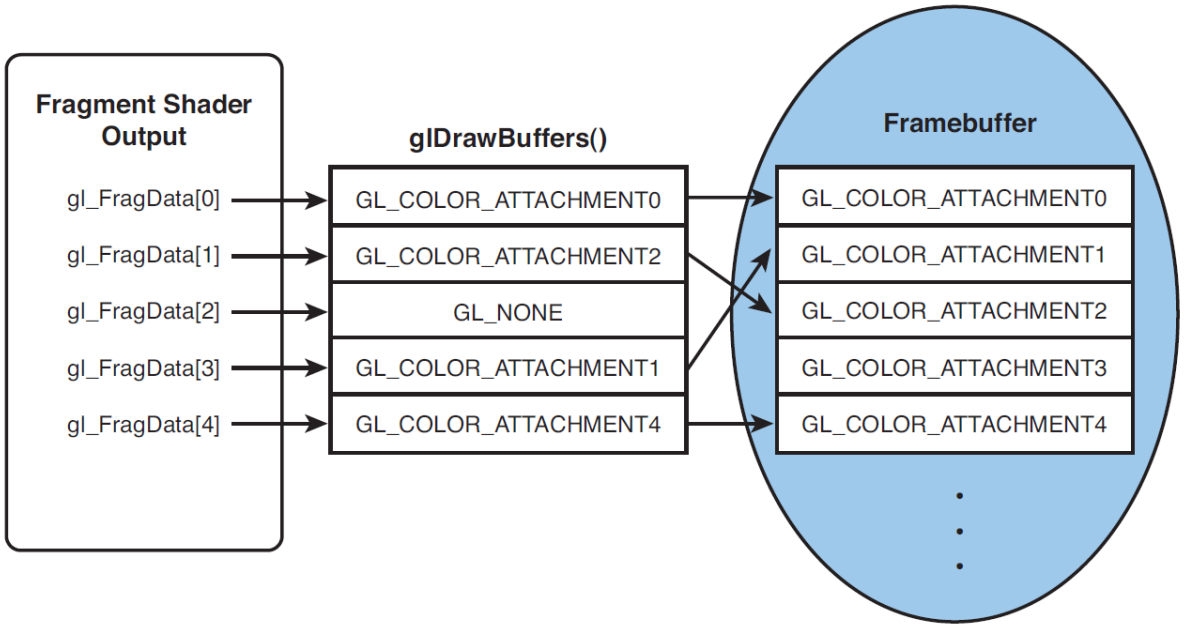
Zamiast funkcji `glBindFragDataLocation` możemy skorzystać z kwalifikatora formatu `layout(location = n)`, który umieszczamy przy zmiennej wyjściowej shadera fragmentów.

Następnie musimy powiedzieć OpenGL dokąd dane z shadera mają być wysłane. Domyślnie OpenGL wysyła tylko jedną wartość koloru do wiązania o numerze 0.

Do określenia, do którego bufora koloru mają trafiać poszczególne kolory wyjściowe shadera fragmentów służy funkcja:

```
void glDrawBuffers(GLsizei n, const enum *bufs);
```

`n` – liczba buforów w `bufs`, zaś `bufs` – tablica ze stałymi określającymi bufory: `GL_NONE`, `GL_COLOR_ATTACHMENTi`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`. Wartości poza `GL_NONE` nie mogą się w `bufs` powtarzać.



Do określenia tylko jednego bufora koloru, do którego ma trafiać kolor wyjściowy shadera fragmentów możemy użyć:

```
void glDrawBuffer(GLenum buf);
```

`buf` – bufor koloru, w którym zapisujemy: `GL_NONE`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, `GL_COLOR_ATTACHMENTi`.

Operacje na buforach

Czyszczenie poszczególnych buforów aktualnie dowiązanego obiektu FBO:

```
void glClearBufferiv(GLenum buffer, GLint drawbuffer, const GLint *value
);
void glClearBufferuiv(GLenum buffer, GLint drawbuffer, const GLuint *
value);
void glClearBufferfv(GLenum buffer, GLint drawbuffer, const GLfloat *
value);
```

`buffer` i `drawbuffer` określają bufor do czyszczenia, a `value` wartość jaką czyścić ten bufor.

Jeśli `buffer` przyjmie wartość `GL_COLOR`, to `drawbuffer` przyjmuje jedną z wartości: `0`, `1`, . . . , maksymalna liczba buforów rysowania, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, a `value` jest 4. elementową tablicą.
Jeśli `buffer` przyjmie wartość `GL_DEPTH`, to `drawbuffer` musi być równe `0`, a `value` jest pojedynczą wartością.

Jeśli `buffer` przyjmie wartość `GL_COLOR`, to `drawbuffer` przyjmuje jedną z wartości: `0`, `1`, . . . , maksymalna liczba buforów rysowania, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, a `value` jest 4. elementową tablicą.
Jeśli `buffer` przyjmie wartość `GL_DEPTH`, to `drawbuffer` musi być równe `0`, a `value` jest pojedynczą wartością.

W celu odczytania wartości z bufora aktualnie dowiązanego obiektu FBO najpierw musimy wybrać bufor, z którego chcemy czytać:

```
void glReadBuffer(GLenum src);
```

`src` określa bufor: `GL_NONE`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, `GL_COLOR_ATTACHMENTi`.

Odczytanie wartości z wybranego bufora:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
GLenum format, GLenum type, GLvoid *data);
```

`x`, `y` – współrzędne pierwszego piksela do odczytania (lewy dolny róg obszaru), `width`, `height` – wymiary obszaru do pobrania, `format` – format danych piksela, `type` – typ danych piksela, `data` – tablica na pobrane dane.

Kopiowanie danych z bufora odczytu do bufora zapisu:

```
void glBlitFramebuffer(GLint srcX0, GLint srcY0, GLint srcX1, GLint
srcY1, GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1, GLbitfield
mask, GLenum filter);
```

`srcX0`, `srcY0`, `srcX1`, `srcY1` – prostokąt źródłowy, `dstX0`, `dstY0`, `dstX1`, `dstY1` – prostokąt docelowy, `mask` przyjmuje jedną z wartości: `GL_DEPTH_BUFFER_BIT`, `GL_STENCIL_BUFFER_BIT`, `GL_COLOR_BUFFER_BIT`, zaś argument `filter` przyjmuje wartości `GL_LINEAR`, `GL_NEAREST`, chyba że kopiowane są dane głębi lub szablonowe to wówczas możemy użyć tylko `GL_LINEAR`.

Efekty specjalne HDR

Efekt poświaty tzw. bloom

Jest to artefakt kamery, który pojawia się na jasnych obiektach i powoduje pojawienie się aureoli (ang. halo) lub poświaty. Gdy obiektyw aparatu próbuje skupić się na jasno oświetlonym obiekcie światło odbijające się od obiektu może wydawać się, że ulatnia się poza obiekt.

Ideą stojącą za efektem poświaty jest odtworzenie tego artefaktu, tak aby scena wyglądała jakby była uchwycona przez rzeczywistą kamerę.

Przykład sceny bez efektu poświaty (po lewej) i z dodanym efektem poświaty (po prawej)

Efekt poświaty jest zazwyczaj tworzony za pomocą renderingu HDR, ale jest możliwa reprodukcja tego efektu w formacie bez HDR. Natura tego efektu wymaga bardzo jasnych obiektów w celu otrzymania dobrej jakości. W przypadku obrazów LDR zakres wartości jest niski, prowadząc do jakości, która nie jest tak dobra jak w przypadku wersji z HDR.

W celu osiągnięcia efektu poświaty musimy wykonać następujące kroki:

- Renderujemy scenę do bufora zmiennoprzecinkowego.
- Renderujemy scenę do drugiego bufora zmiennoprzecinkowego, ale zapisujemy w nim jedynie informacje o najjaśniejszych fragmentach, tzn. fragmentach, który luminancja jest powyżej określonego progu.
  (Uwaga. Oba punkty możemy wykonać za pomocą jednego przebiegu renderingu jeśli korzystamy z renderingu do wielu buforów)
- Wykonujemy rozmycie w pionie, np. za pomocą filtru Gaussa, zawartości bufora z informacją o najjaśniejszych fragmentach.
- Wykonujemy rozmycie w poziomie, np. za pomocą filtru Gaussa, zawartości bufora po rozmyciu w pionie.
- Mieszamy bufor z normalnie wyrendorowaną sceną (pierwszy punkt) z buforem otrzymanym po rozmywaniu.

Efekt smugi (ang. streak)

Jeżdżąc np. samochodem możemy zauważyć, że uliczne latarnie czy reflektory innych samochodów mają wokół siebie smugę w kształcie przypominającym gwiazdę. Zjawisko to jest spowodowane wewnętrznymi odbiciami i załamaniami występującymi dzięki mikroskopijnym rysom na powierzchni szkła czy to naszej szyby czy soczewki kamery.

Fizyka jaka stoi za tym efektem jest złożona dlatego na potrzeby renderingu czasu rzeczywistego wykorzystujemy pewną aproksymację, która da nam zadawalający efekt.

Żeby osiągnąć ten efekt musimy stworzyć gwieździsty wzór smug. Do tego celu możemy wykorzystać specyficzny filtr rozmywający, który działa w określonym kierunku tworząc tym samym smugę w tym kierunku. Rozmycia dokonujemy, podobnie jak dla efektu poświaty, na buforze z informacją o najjaśniejszych fragmentach.

Dla każdego kierunku tworzymy osobny obraz. Mając obrazy dla wszystkich kierunków tworzymy ostateczny obraz z gwieździstym wzorem przez uśrednienie kolorów z tych obrazów.

Mając obraz ze smugami możemy go połączyć z obrazem wyrenderowanej sceny oraz efektu poświaty.

Do filtracji możemy użyć następującego sposobu. Niech `samplesDir` będzie tablicą (4. elementową) ze współrzędnymi próbek w określonym kierunku, `tex` będzie teksturą, którą rozmywamy, a `texCoord` współrzędnymi tekstury fragmentu, który przetwarzamy. Ponadto `blurFactor` i `blurOffset` będą parametrami kontrolującymi efekt.

Na początku przyjmujemy `color = vec4(0, 0, 0, 0)`.
Następnie dla `i = 0, 1, 2, 3` wykonujemy obliczenia:

- `blur = pow( blurFactor, i )`
- ```
  sample = texture( tex,
                    texCoord + offsetFactor * samplesDir[i] )
  ```
- `color += blur * sample`

Na koniec w `color` mamy kolor wyjściowy dla fragmentu.
  ```

Dynamiczne mapowanie tonów

Ludzkie oko nie adaptuje się cały czas do drastycznych zmian w jasności. Oko potrzebuje nieco czasu aby zaadaptować się do światła słonecznego po spędzeniu znacznego czasu w ciemności i vice versa.

Do symulacji tego efektu możemy wykorzystać dynamiczne mapowanie tonów. Można je wykonać na różne sposoby.

Jednym z podejść jest użycie, zamiast średniej luminancji sceny, zaadaptowanej luminancji sceny, którą możemy obliczyć następująco:

$$L_a^{new} = L_a + (L_{w,H} - L_a)\left(1 - \exp\left(-\frac{T}{\tau}\right)\right),$$

gdzie L_a – wartość poprzedniej zaadaptowanej luminancji sceny, T – czas od ostatniej klatki, τ – stała adaptacji.

Stała adaptacji jest różna dla pręcików (0.4 s) i czopków (0.1 s). Zatem τ obliczane jest następująco:

$$\tau = 0.4\sigma + 0.1(1 - \sigma),$$

gdzie σ – wrażliwość pręcików:

$$\sigma = \frac{0.04}{0.04 + L_{w,H}}.$$