Programowanie warstwy wizualnej gry

Krzysztof Gdawiec

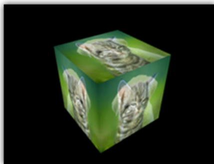
UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

Bufor szablonowy

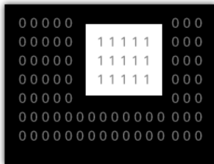
Bufor szablonowy (ang. stencil buffer) umożliwia utworzenie specjalnego szablonu (matrycy) określającego obszar w buforze koloru, który będzie używany do renderingu.

Bufor szablonowy zawiera dane w postaci liczb całkowitych. Liczba bitów przypadająca na jeden element bufora zależy od implementacji, ale nie może być mniejsza niż 1.

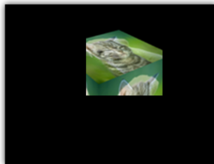
Bufor szablonowy stosowany jest do osiągnięcia różnych efektów. Przykładami takich efektów są m.in. efekt odbicia, technika CSG.



Color buffer without stencil test



Stencil buffer



Color buffer with stencil test

Zawartość bufora szablonowego można wypełnić (czyścić) stałą wartością za pomocą funkcji `glClear` z parametrem `GL_STENCIL_BUFFER_BIT`. Domyślnie bufor wypełniany jest zerami, ale wartość tę można zmienić za pomocą funkcji

```
void glClearStencil(GLint s);
```

Zawartość bufora szablonowego można wypełnić (czyścić) stałą wartością za pomocą funkcji `glClear` z parametrem `GL_STENCIL_BUFFER_BIT`. Domyślnie bufor wypełniany jest zerami, ale wartość tę można zmienić za pomocą funkcji

```
void glClearStencil(GLint s);
```

Działanie bufora szablonowego reguluje kilka funkcji. Pierwszą z nich jest:

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
void glStencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint
mask);
```

która ustala test szablonu.

Parametr `func` wskazuje rodzaj zastosowanej funkcji testującej (s – wartość znajdująca się w buforze):

- `GL_NEVER` – wartość testu zawsze negatywna,
- `GL_LESS` – wartość testu pozytywna jeśli `ref` $< s$,
- `GL_LEQUAL` – wartość testu pozytywna jeśli `ref` $\leqslant s$,
- `GL_GREATER` – wartość testu pozytywna jeśli `ref` $> s$,
- `GL_GEQUAL` – wartość testu pozytywna jeśli `ref` $\geqslant s$,
- `GL_EQUAL` – wartość testu pozytywna jeśli `ref` $= s$,
- `GL_NOTEQUAL` – wartość testu pozytywna jeśli `ref` $\neq s$,
- `GL_ALWAYS` – wartość testu zawsze pozytywna.

Wartość parametru `ref` obcinana jest do przedziału $[0, 2^n - 1]$, gdzie n jest liczbą bitów bufora szablonowego.

Parametr `mask` określa, dla których bitów wartości referencyjnej `ref` i wartości w buforze wykonywany jest test szablonu. Operacja ta jest realizowana za pomocą iloczynu logicznego testowanych wartości z wartością maski.

Parametr `face` określa stronę: `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`.

Parametr `mask` określa, dla których bitów wartości referencyjnej `ref` i wartości w buforze wykonywany jest test szablonu. Operacja ta jest realizowana za pomocą iloczynu logicznego testowanych wartości z wartością maski.

Parametr `face` określa stronę: `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`.

Do określania operacji na buforze szablonowym służy funkcja:

```
void glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass);
void glStencilOpSeparate(GLenum face, GLenum sfail, GLenum dpfail,
GLenum dppass);
```

Każdy parametr odpowiada za operację w innej sytuacji.

- parametr `sfail` odpowiada za przypadek negatywnego wyniku testu szablonowego,
- parametr `dpfail` odpowiada za przypadek pozytywnego wyniku testu szablonowego przy negatywnym wyniku testu bufora głębi,
- parametr `dppass` odpowiada za przypadek pozytywnego wyniku testu bufora szablonowego i bufora głębi (także gdy bufor głębi jest wyłączony).

Każdy z parametrów `sfail`, `dpfail`, `dppass` może przyjąć jedną z wartości:

- `GL_KEEP` – wartość bufora szablonowego nie jest zmieniana,
- `GL_ZERO` – wartość bufora szablonowego jest zerowana,
- `GL_REPLACE` – wartość bufora szablonowego jest zmieniana na wartość referencyjną określoną w parametrze `ref` funkcji `glStencilFunc`,
- `GL_INCR` – wartość bufora szablonowego jest zwiększana o 1; w przypadku nadmiaru przypisywana jest wartość maksymalna,
- `GL_DECR` – wartość bufora szablonowego jest zmniejszana o 1; w przypadku niedomiaru przypisywana jest wartość 0,

- GL_INVERT – wartość bufora szablonowego jest negowana (negowane są bity),
- GL_DECR_WRAP – wartość bufora szablonowego jest zmniejszana o 1; w przypadku niedomiaru przypisywana jest wartość maksymalna,
- GL_INCR_WRAP – wartość bufora szablonowego jest zwiększana o 1; w przypadku nadmiaru przypisywana jest wartość 0.

Parametr face określa stronę: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.

Domyślnie test szablonowy jest wyłączony. Włączamy go za pomocą `glEnable` z parametrem `GL_STENCIL_TEST`.

Możemy też maskować zapis do bufora szablonowego (1 – zapis na danym bicie jest możliwy, 0 – brak zapisu):

```
void glStencilMask(GLuint mask);
void glStencilMaskSeparate(GLenum face, GLuint mask);
```

Instancjonowanie

Zdarzają się sytuacje, gdy ten sam obiekt chcemy narysować wiele razy, np. flota statków, źdźbła trawy.

W takiej sytuacji moglibyśmy w pętli przejść po poszczególnych obiektach i rysować każdy z nich za pomocą np. `glDrawArrays`.

Przy takim podejściu może się okazać, że rendering pojedynczego obiektu jest krótki, więc aplikacja będzie poświęcać więcej czasu na wysyłanie poleceń do OpenGL niż na renderingu.

W OpenGL problem ten możemy rozwiązać poprzez instancjonowanie czyli rysowanie wielu instancji obiektu.

W instancjonowaniu w jednym wywołaniu funkcji wskazujemy zamiar wyświetlenia wielu kopii tej samej geometrii. Do tego celu mamy funkcje:

```
void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count,
GLsizei instancecount);
void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type,
const void* indices, GLsizei instancecount);
```

Argumenty `mode`, `first`, `count`, `type`, `indices` mają takie samo znaczenie jak w przypadku funkcji `glDrawArrays` i `glDrawElements`. Argument `instancecount` oznacza liczbę kopii geometrii do wyrysowania.

Jeśli `instancecount = 1`, to rezultat działania tych funkcji jest taki sam jak `glDrawArrays` i `glDrawElements`.

Gdyby funkcje `glDrawArraysInstanced`, `glDrawElementsInstanced` rysowały kopię tych samych wierzchołków nie byłoby to zbyt użyteczne.

Tym co czyni instancjonowanie użytecznym jest specjalna, wbudowana w GLSL zmienna `gl_InstanceID`, która jest typu `int`.

Przy wysyłaniu pierwszej kopii wierzchołków `gl_InstanceID` przyjmuje wartość 0. Każda następna kopia geometrii zwiększa wartość zmiennej aż do osiągnięcia `instancecount - 1`.

Używając zmiennej `gl_InstanceID` w shaderze wierzchołków możemy zmienić położenie każdej z kopii. Zatem inteligentne wykorzystanie tej zmiennej daje dużą elastyczność podczas renderowania.

Powiedzmy, że chcemy każdą kopię narysować innym kolorem. Do tego celu możemy wykorzystać zmienną `gl_InstanceID`, ale w OpenGL istnieje specjalny mechanizm nazywany tablicą kopii.

W celu użycia takiej tablicy definiujemy zmienną wejściową shadera wierzchołków (atrybut). Pobieramy jego lokalizację i umieszczamy dane w obiekcie VBO związanym z tym atrybutem.

Standardowo wartości atrybutów zmieniane są dla każdego wierzchołka. Aby wymusić pobieranie wartości raz na kopię używamy funkcji:

```
void glVertexAttribDivisor(GLuint location, GLuint divisor);
```

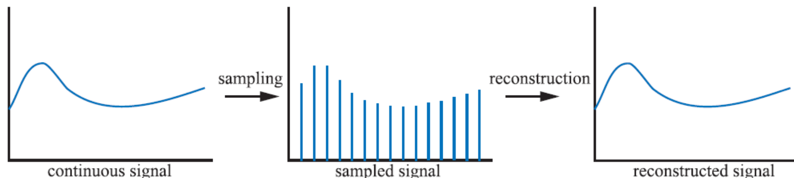
`location` to lokalizacja atrybutu, `divisor` to wartość co ile kopii ma być pobierana nowa wartość z tablicy.

```
#version 450

in vec4 position;
in vec4 color;

out vec4 vColor;

uniform mat4 mvp;

void main()
{
  gl_Position = mvp * position;
  vColor = color;
}
```

`glVertexAttribDivisor(color_location, 1);` – kolor zmieni się dla każdej kopii

`glVertexAttribDivisor(color_location, 2);` – kolor zmieni się co drugą kopię

Antialiasing

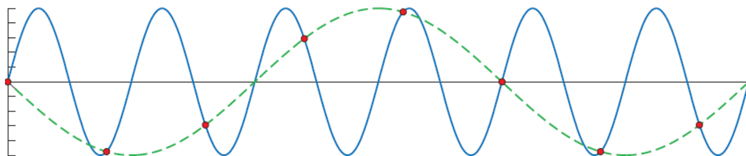


Celem procesu próbkowania (ang. sampling) jest przedstawienie informacji w sposób cyfrowy. W tym celu ilość informacji jest redukowana.
Spróbkowany sygnał musi być zrekonstruowany w celu odzyskania oryginalnego sygnału. Robimy to za pomocą filtracji sygnału spróbkowanego.

Gdziekolwiek wykonujemy próbkowanie może pojawić się aliasing. Jest to niechciany artefakt, z którym musimy walczyć, aby otrzymać dobry obraz.

Standardowym przykładem aliasingu w grafice komputerowej są poszarpane (ang. jagged) krawędzie.

Aliasing pojawia się kiedy sygnał próbkowany jest ze zbyt małą częstotliwością. Wówczas spróbkowany sygnał wygląda jakby miał mniejszą częstotliwość niż oryginalny sygnał.



Oryginalny sygnał na niebiesko, spróbkowany sygnał na zielono, czerwone punkty to punkty próbkowania.

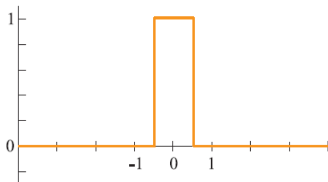
Aby dobrze spróbkować sygnał (tzn. tak, aby była możliwa rekonstrukcja oryginalnego sygnału z próbek) częstotliwość próbkowania musi być większa niż dwukrotność maksymalnej częstotliwości próbkowanego sygnału.

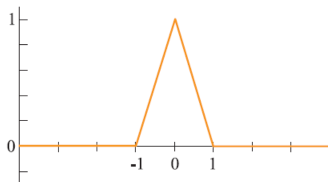
Częstotliwość graniczną nazywamy granicą Nyquista.

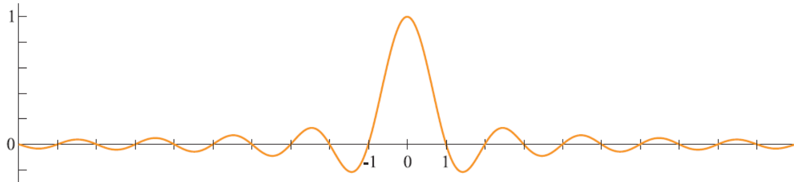
„Maksymalna częstotliwość" implikuje, że sygnał musi mieć ograniczone pasmo, tzn. że nie ma częstotliwości powyżej pewnej granicy.

Rekonstrukcja

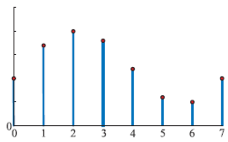
Rekonstrukcji dokonujemy za pomocą filtrów.

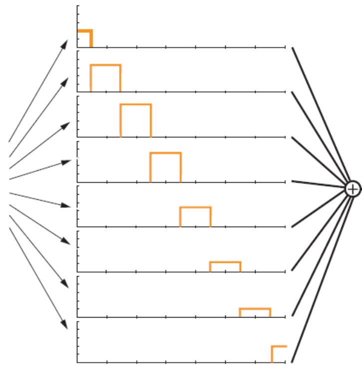


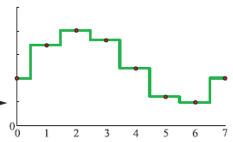


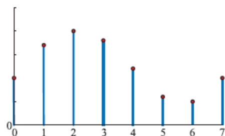


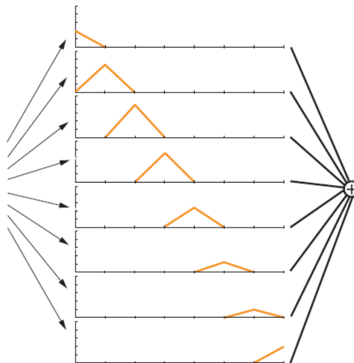
Box filter, tent filter, sinc filter ($sinc(x) = \sin(\pi x)/(\pi x)$, gdy $x \neq 0$ oraz $sinc(0) = 1$).

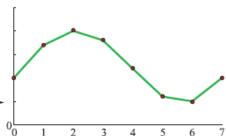

0
0 1 2 3 4 5 6 7

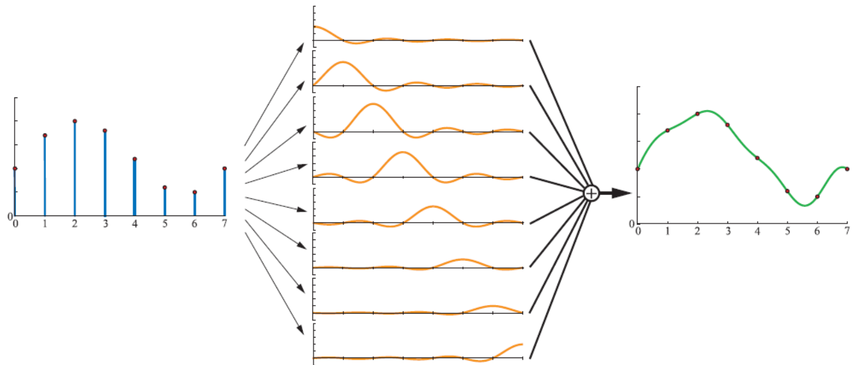



0
0 1 2 3 4 5 6 7


0
0
1
2
3
4
5
6
7




0
0
1
2
3
4
5
6
7


0
0 1 2 3 4 5 6 7
0
0 1 2 3 4 5 6 7

Przepróbkowanie

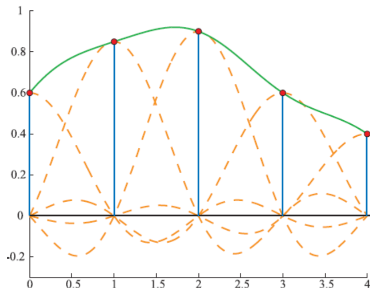
Przepróbkowanie jest używane do powiększania lub pomniejszania próbkowanego sygnału.

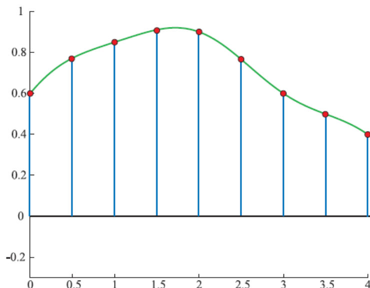
Załóżmy, że mamy oryginalne punkty próbkowania, które znajdują się w liczbach całkowitych $(0, 1, 2, \ldots)$. Ponadto załóżmy, że po przepróbkowaniu chcemy żeby nowe punkty próbkowania były rozłożone równomiernie w odległości a pomiędzy punktami.

Jeśli $a > 1$, to miejsce ma pomniejszenie (downsampling). Jeśli $a < 1$, to miejsce ma powiększenie (upsampling).

Załóżmy, że chcemy wykonać powiększenie i że mamy sygnał zrekonstruowany.

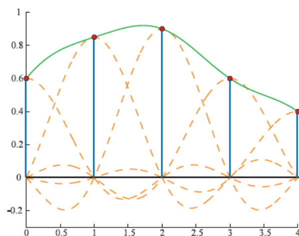
Wystarczy, że poddamy sygnał zrekonstruowany próbkowaniu o zadanej odległości pomiędzy punktami próbkowania.

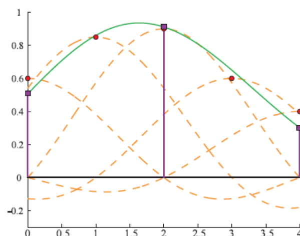




Załóżmy, że chcemy wykonać pomniejszenie i że mamy sygnał zrekonstruowany.

Używając punktów próbkowania oraz filtra $sinc(x/a)$ rekonstruujemy sygnał, który następnie poddajemy próbkowaniu o odległości a pomiędzy punktami próbkowania.



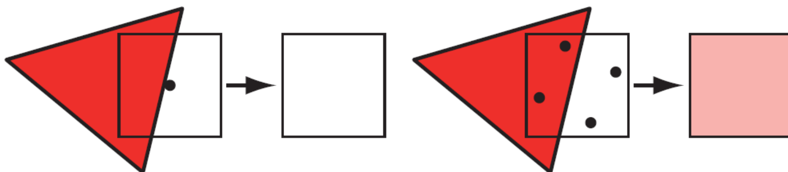


Przekładając to na obrazy. Proces jest podobny do rozmycia obrazu i późniejszego zmniejszenia jego rozdzielczości.

Antialiasing na ekranie (ang. screen-based antialiasing)

Algorytmy tego typu operują tylko na próbkach wyjściowych potoku i nie potrzebują żadnej wiedzy nt. obiektów, które renderujemy.

Jednym z powodów powstawania aliasów jest zbyt mały współczynnik próbkowania.



Ogólną strategią w algorytmach antialiasingu na ekranie jest użycie wzoru próbkowania na ekranie i obliczenie wynikowego koloru:

$$p(x, y) = \sum_{i=1}^{n} w_i c(i, x, y),$$

gdzie n – liczba próbek na piksel, c – funkcja próbkująca kolor, $w_i \in [0, 1]$ – waga próbki, $\sum w_i = 1$.

Full-scene antialiasing (FSAA)

Najprostsza metoda, która renderuje scenę w wyższej rozdzielczości, a następnie uśrednia sąsiadujące próbki. Np. załóżmy, że chcemy obraz o rozdzielczości 1000×800. Renderujemy go w rozdzielczości 2000×1600 czyli będziemy mieli 4 próbki na piksel.

Metoda jest bardzo prosta, ale równocześnie bardzo kosztowna. Dla każdej z próbek musimy wykonać pełne obliczenia cieniowania, wykonać test głębi itp.

Accumulation buffer

Zamiast jednego dużego bufora pozaekranowego używamy bufora, który ma taką samą rozdzielczość jak wynikowy obraz.

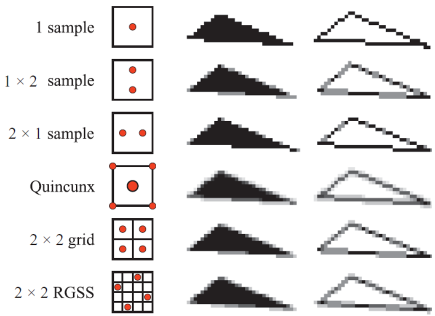
Jeśli chcemy otrzymać próbkowanie 2×2 generujemy 4 obrazy z widokiem przesuniętym o pół piksela na ekranie w kierunku x i y. Obrazy te są sumowane w buforze akumulacyjnym. Po renderowaniu obraz jest uśredniany (dla 2×2 dzielimy przez 4), a następnie wyświetlany.

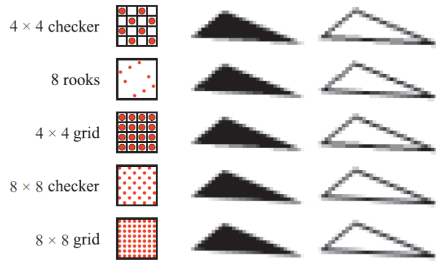
Ponieważ scena renderowana jest kilkukrotnie metoda jest kosztowna.

Rotated Grid Supersampling (RGSS)

Wzór próbkowania nie jest równomierny. Zamiast tego używa obróconego kwadrata, np. $(0, 0.25)$, $(0.5, 0)$, $(0.75, 0.5)$, $(0.25, 0.75)$.

Taki wzór daje więcej poziomów antialiasingu dla prawie pionowych lub poziomych krawędzi, które to najczęściej wymagają poprawy.


1 sample
1 × 2 sample
2 × 1 sample
Quincunx
2 × 2 grid
2 × 2 RGSS


4 × 4 checker
8 rooks
4 × 4 grid
8 × 8 checker
8 × 8 grid

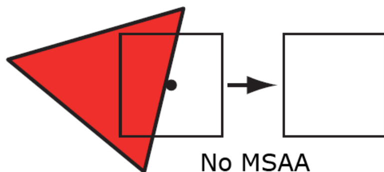
Multisample antialiasing (MSAA)

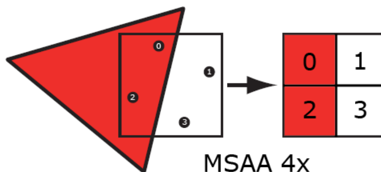
Algorytmy z tej grupy zmniejszają koszt obliczeniowy w stosunku do poprzednich metod poprzez dzielenie wyników obliczeń między próbkami.

Na pojedynczy piksel w wynikowym obrazie przypada określona liczba próbek (sub-pikseli). Bufor koloru i głębi muszą mieć odpowiedni rozmiar, aby przechować dane dla próbek.

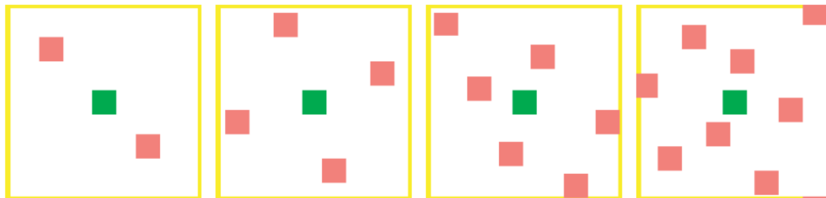
Każdy sub-piksel może przechowywać indeks do fragmentu, z którym jest związany. Dla każdego fragmentu zapamiętujemy tablicę przechowującą kolor i wartość głębi.

Podczas rasteryzacji prymitywu sprawdzane jest pokrycie próbek przez prymityw. Jeśli choć jedna próbka jest pokryta, to wykonywane są obliczenia (uruchamiany jest shader) w środku piksela, a wynik zapamiętywany jest w pokrytych próbkach.
Na końcu kolory próbek dla każdego piksela są uśredniane tworząc wynikowy kolor.





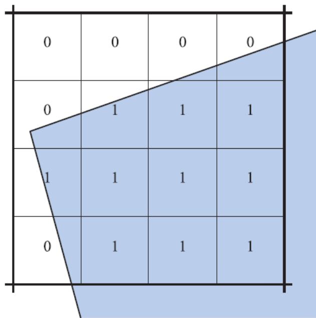
Przykładowe wzory próbkowania. Próbki, dla których wykonujemy uruchomienie shadera oznaczone są na zielono, zaś na czerwono próbki, dla których sprawdzamy pokrycie.



Bufor A (ang. A-buffer)

Bufor A wykorzystuje fakt rozdzielenia pokrycia od koloru cieniowania i wartości głębi.

Każdy renderowany wielokąt tworzy maskę pokrycia dla każdej komórki siatki na ekranie.



Maska: 0000 0111 1111 0111

Podobnie jak w MSAA kolor dla wielokąta związanego z maską pokrycia jest obliczany tylko raz w środku fragmentu i dzielony jest przez próbki.

Kiedy wszystkie wielokąty są umieszczone w buforze A obliczany jest kolor dla każdego z fragmentów. Na podstawie maski pokrycia obliczana jest widzialność fragmentu. Następnie wartość ta mnożona jest przez kolor i jest dodawana do wynikowego koloru.

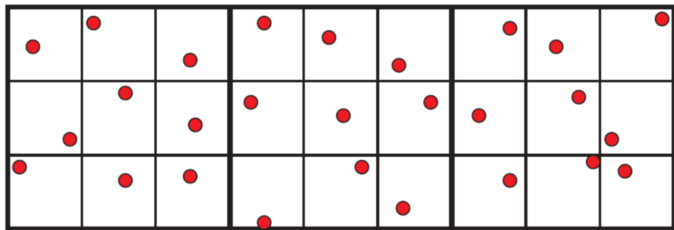
Próbkowanie stochastyczne

Scena może składać się z obiektów, które są dowolnie małe, tzn. żaden sposób próbkowania nie jest w stanie ich dobrze uchwycić. W związku z tym regularne sposoby próbkowania będą tworzyły aliasing.

Jednym z podejść, aby uniknąć aliasingu jest losowy rozkład próbek w pikselu, z różnym wzorem próbkowania dla każdego piksela. Takie podejście nazywamy próbkowaniem stochastycznym.

Przykładem takiego próbkowania jest rozsiewanie (ang. jittering).

Załóżmy, że chcemy użyć n próbek. Dzielimy obszar piksela na n regionów o równej powierzchni. Każdą próbkę umieszczamy w losowy sposób w jednym z regionów.



Innym przykładem próbkowania stochastycznego jest wzór N wież (ang. N-rooks pattern). n próbek jest rozmieszczonych na siatce $n \times n$ tak, że jedna próbka przypada na wiersz i kolumnę.

Antialiasing w OpenGL

W OpenGL mamy możliwość skorzystania z MSAA.

W tym celu musimy stworzyć kontekst OpenGL z odpowiednim buforem ramki. W bibliotece GLFW przed utworzeniem kontekstu dodajemy wskazówkę, która mówi o liczbie próbek:

```
glfwWindowHint( GLFW_SAMPLES, noSamples );
```

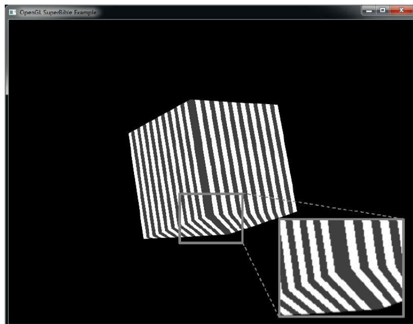
`noSamples` to liczba próbek.

Do włączania/wyłączania MSAA możemy później użyć `glEnable`/`glDisable` z parametrem `GL_MULTISAMPLE`.

Położenie próbek zależy od implementacji OpenGL.

Cieniowanie próbek

Uruchomienie shadera raz na fragment nie zawsze da dobry rezultat. Jeśli shader tworzy wyjście o wysokiej częstotliwości, to ponownie pojawia się aliasing.



W celu zmniejszenia aliasingu możemy użyć cieniowania próbek (ang. sample-rate shading).

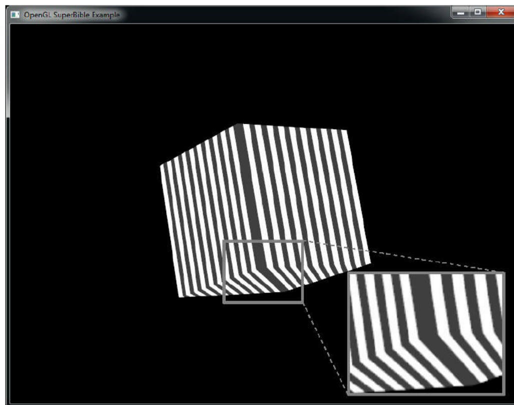
Do włączenia cieniowania próbek używamy `glEnable` z parametrem `GL_SAMPLE_SHADING`. Następnie musimy powiedzieć, dla której części próbek OpenGL powinien uruchamiać shader:

```
void glMinSampleShading(GLfloat value);
```

`value = 1` określa, że dla każdej pokrytej próbki uruchamiamy shader, `value = 0` określa, że ignorujemy cieniowanie próbek.

Dla których próbek zostanie uruchomiony shader i algorytm wyboru podzbioru próbek zależny jest od implementacji OpenGL.

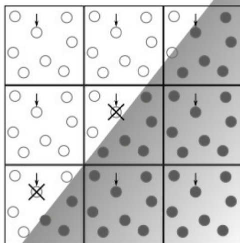
Cieniowanie próbek włączone z `value = 1`.



Próbkowanie centroidalne

Shader fragmentów interpoluje wartości zmiennych względem próbki, która leży najbliżej środka.

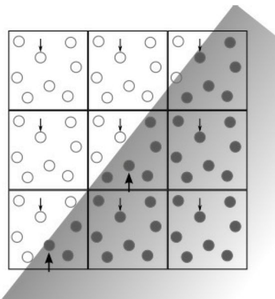
Jeśli znajdujemy się w środku trójkąta, to nie ma problemu. Problemy zaczynają się na krawędziach.



Rozwiązaniem tego problemu jest oznaczenie zmiennej wejściowej shadera fragmentu kwalifikatorem `centroid`, np.

```
centroid in vec2 texCoord;
```

W takim przypadku wartość jest interpolowana do punktu znajdującego się w pikselu i renderowanym prymitywie lub próbki wewnątrz prymitywu.



Dlaczego kwalifikator `centroid` nie jest domyślnym kwalifikatorem dla zmiennych wejściowych shadera fragmentów?

Najważniejszą wadą jest niedokładne obliczanie gradientów dla wejścia shadera fragmentów. Jest to spowodowane tym, że do obliczeń gradientów używa się dyskretyzacji pochodnych, a przy próbkowaniu centroidalnym odległości pomiędzy sąsiednimi pikselami nie są takie same.

Tekstury wielopróbkowe

Do obiektów FBO możemy dołączać tekstury wielopróbkowe.

W tym celu tworzymy obiekt tekstury. Następnie tworzymy teksturę wielopróbkową:

```
void glTextureStorage2DMultisample(GLuint texture, GLsizei samples,
GLenum internalformat, GLsizei width, GLsizei height, GLboolean
fixedsamplelocations);
void glTextureStorage3DMultisample(GLuint texture, GLsizei samples,
GLenum internalformat, GLsizei width, GLsizei height, GLsizei depth,
GLboolean fixedsamplelocations);
```

`texture` to identyfikator obiektu tekstury, `samples` to liczba próbek w teksturze, `internalformat` to format wewnętrzny tekstury, `width`, `height`, `depth` to wymiary tekstury, `fixedsamplelocations` określa czy używać identycznych lokalizacji próbek dla wszystkich tekseli.

Dołączenie tak stworzonej tekstury do obiektu FBO odbywa się standardowo za pomocą funkcji `glFramebufferTexture`.

Ograniczenia tekstur wielopróbkowych:

- nie istnieją wersje jedno- i trójwymiarowe,
- nie posiadają mipmap,
- nie obsługują filtrowania.

W shaderze do tekstur wielopróbkowych odwołujemy się za pomocą samplerów: `sampler2DMS`, `sampler2DMSArray`, `isampler2DMS`, `usampler2DMS`, `isampler2DMSArray`, `usampler2DMSArray`.

Do pobierania teksela używamy:

```
vec4 texelFetch(sampler2DMS sampler, ivec2 P, int sample);
vec4 texelFetch(gsampler2DMSArray sampler, ivec3 P, int sample);
```