Programowanie warstwy wizualnej gry

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

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`.

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.

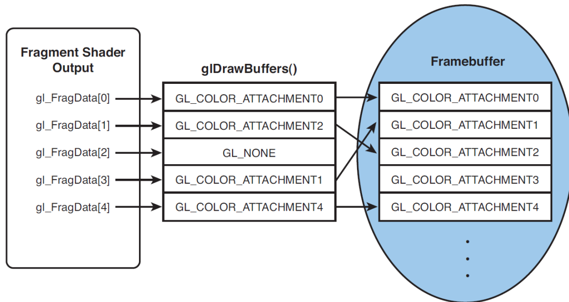
Zamiast funkcji `glBindFragDataLocation` możemy skorzystać z kwalifikatora formatu `layout(location = n)`, który umieszczamy przy zmiennej wyjściowej shadera fragmentów.

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`.

Selekcja obiektu

Bardzo często potrzebujemy wybrać jakiś obiekt na scenie za pomocą myszki. W przypadku 3D nie jest to zadanie trywialne.

Do selekcji obiektu w 3D istnieją różne algorytmy, np.

- selekcja z użyciem bufora głębi,
- selekcja na podstawie koloru,
- selekcja na podstawie przecięcia z promieniem oka.

Selekcja z użyciem bufora głębi

Potrzebować będziemy sposobu na przeliczenie punktu w oknie na punkt w przestrzeni 3D.

Do tego celu w bibliotece GLM służy funkcja:

```
vec3 unProject(vec3 win, mat4 modelView, mat4 projection, vec4 viewport)
;
```

`win` – współrzędne w przestrzeni okna, `modelView` – macierz model-widok, `projection` – macierz rzutowania, `viewport` – informacje o obszarze renderingu, tzn. lewy dolny róg obszaru oraz szerokość i wysokość.

Funkcja `unProject` oblicza punkt w następujący sposób:

$$(PVM)^{-1} \begin{pmatrix} \frac{2(w_x - v_0)}{v_2} - 1 \\ \frac{2(w_y - v_1)}{v_3} - 1 \\ 2w_z - 1 \\ 1 \end{pmatrix}$$

gdzie P – macierz rzutowania, V – macierz widoku, M – macierz modelu, (w_x, w_y, w_z) – współrzędne w przestrzeni okna, (v_0, v_1, v_2, v_3) – informacje o obszarze renderingu.

- Włączamy testowanie głębi.
- Po kliknięciu myszką odczytujemy wartość z bufora głębi znajdującą się pod kursorem – funkcja `glReadPixels`.
- Przeliczamy punkt z przestrzeni okna (ze współrzędną z równą odczytanej wartości głębi) do przestrzeni modelu – funkcja `unProject`.
- Znajdujemy obiekt znajdujący się najbliżej punktu wyznaczonego w poprzednim punkcie.

Selekcja na podstawie koloru

- Każdy obiekt na scenie ma przypisany unikatowy kolor.
- Renderujemy scenę do dwóch buforów. Pierwszy zawiera normalnie wyrenderowaną scenę – bufor koloru. W drugim renderujemy obiekty używając unikatowych kolorów (nie obliczamy oświetlenia, nie teksturujemy itp.) – bufor selekcji.
- Po kliknięciu myszką z bufora selekcji odczytujemy kolor pod kursorem – funkcja `glReadPixels`.
- Znajdujemy obiekt, którego unikatowy kolor równy jest odczytanemu kolorowi.

Selekcja na podstawie przecięcia z promieniem oka

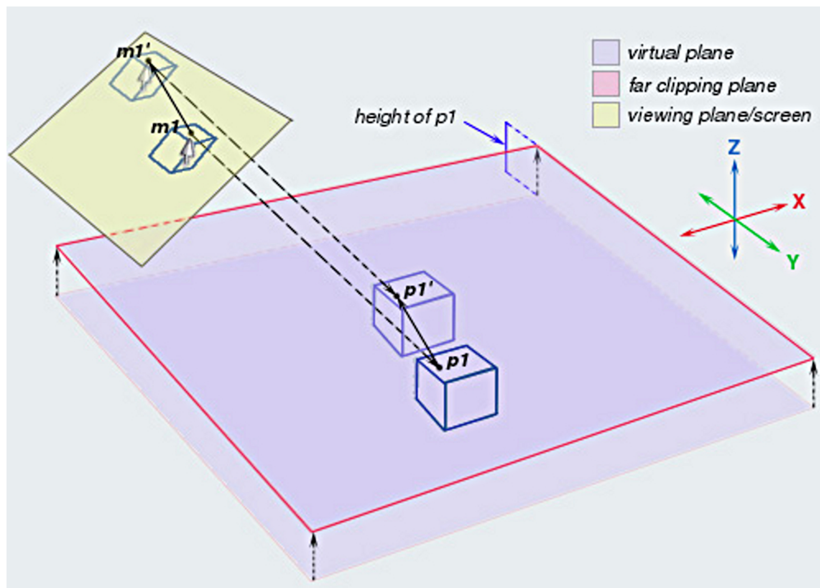
- Po kliknięciu myszką przeliczamy punkt (w przestrzeni okna) pod kursorem na przestrzeń modelu dla dwóch wartości współrzędnej z: 0 i 1. Otrzymujemy p_0 i p_1.
- Tworzymy promień r, którego początkiem jest lokalizacja kamery, a wektor kierunku jest równy $\frac{p_1 - p_0}{||p_1 - p_0||}$.
- Sprawdzamy przecięcie każdego z obiektów na scenie z promieniem r i wybieramy ten, którego punkt przecięcia leży najbliżej kamery.

Sprawdzanie przecięcia całego obiektu z promieniem może być kosztowne. Dlatego możemy zastosować uproszczenie w postaci sprawdzania przecięcia promienia z prostopadłościanem ograniczającym model. Metoda będzie szybsza, ale mniej precyzyjna.

Poruszanie obiektem po scenie za pomocą myszki

Idea translacji bazującej na wirtualnej płaszczyźnie (w skrócie):

- Wybieramy obiekt i przesuwamy wirtualną płaszczyznę (płaszczyzna, której nie rysujemy na scenie, najczęściej wyrównana do jednej z płaszczyzn XY, XZ, YZ) do punktu, który został wybrany na obiekcie.
- Przy ruchu kursora wystrzeliwujemy promień z punktu, w którym znajduje się kursor w głąb sceny 3D.
- Testujemy przecięcie tego promienia z wirtualną płaszczyzną.
- Przesuwamy obiekt o wektor wyznaczony przez obecny punkt przecięcia oraz poprzedni punkt przecięcia.


virtual plane
far clipping plane
viewing plane/screen
height of p1
m1'
m1
p1'
p1
Z
X
Y

Selekcja obiektu:

- wystrzel promień z punktu m_1 (punkt okna), który przecina obiekt; otrzymujemy pozycję piksela w punkcie przecięcia p_1 (punkt w 3D),
- wyrównaj wirtualną płaszczyznę do jednej z płaszczyzn (XY, XZ, YZ),
- przesuń wirtualną płaszczyznę na wysokość wyznaczoną przez punkt p_1.

Ruch obiektem:

- wykonaj test przecięcia promień-płaszczyzna – wystrzel promień z punktu m_1' (punkt okna) tak żeby przeciął wirtualną płaszczyznę; otrzymujemy w ten sposób punkt p_1',
- przesuń obiekt o wektor $p_1' - p_1$,
- niech $p_1 = p_1'$ (przygotowanie do następnego ruchu myszką)

Przy takim podejściu występuje jeden szczególny przypadek, gdy płaszczyzna widzenia jest prostopadła do wirtualnej płaszczyzny. Wówczas test przecięcia promień-płaszczyzna zakończy się niepowodzeniem.

Przy takim podejściu występuje jeden szczególny przypadek, gdy płaszczyzna widzenia jest prostopadła do wirtualnej płaszczyzny. Wówczas test przecięcia promień-płaszczyzna zakończy się niepowodzeniem.

Rozwiązaniem tego problemu może być wyrównanie wirtualnej płaszczyzny do płaszczyzny, która nie będzie do niej prostopadła. Do tego celu możemy przeprowadzić test przecięcia promień-płaszczyzna dla trzech możliwych płaszczyzn (XY, XZ, YZ) i wybrać tą, która przeszła ten test.

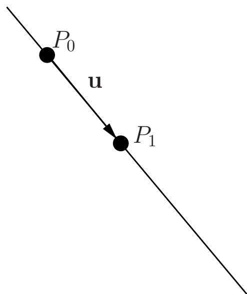
Test przecięcia promień-płaszczyzna

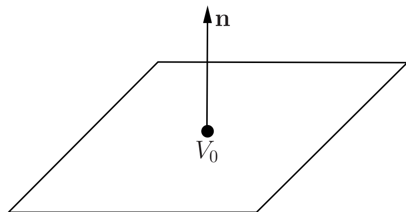
Promień dany jest równaniem parametrycznym:

$$P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathrm{u},$$

gdzie P_0 – punkt początkowy, P_1 – punkt na promieniu, u – wektor kierunku, $s \in [0, +\infty)$.

Płaszczyzna jest dana za pomocą punktu V_0 należącego do tej płaszczyzny oraz wektora normalnego do niej n.



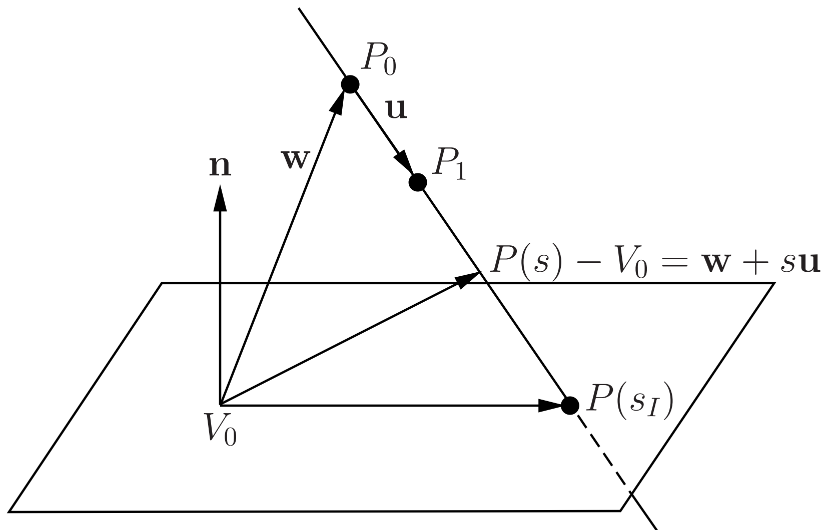


- Obliczamy:

$$a = \mathrm{n} \cdot \mathrm{u},$$

gdzie $\cdot$ to iloczyn skalarny.

- Jeśli $a = 0$, to promień i płaszczyzna są równoległe lub promień leży na płaszczyźnie. Brak przecięcia.
- Jeśli $a \neq 0$, to promień i płaszczyzna nie są równoległe, a zatem posiadają jeden punkt przecięcia $P(s_I)$.


P_0
$\mathbf{u}$
$\mathbf{w}$
$\mathbf{n}$
P_1
$P(s) - V_0 = \mathbf{w} + s\mathbf{u}$
$P(s_I)$
V_0

- W punkcie przecięcia wektor

$$P(s) - V_0 = (P_0 - V_0) + s\mathrm{u} = \mathrm{w} + s\mathrm{u}$$

jest prostopadły do wektora n. Zatem

$$\mathrm{n} \cdot (\mathrm{w} + s_I \mathrm{u}) = 0.$$

Rozwiązując to równanie otrzymujemy

$$s_I = \frac{-\mathrm{n} \cdot \mathrm{w}}{\mathrm{n} \cdot \mathrm{u}} = \frac{-\mathrm{n} \cdot \mathrm{w}}{a}$$

- Jeśli $s_I < 0$, to przecięcie nastąpiło w kierunku odwrotnym do danego wektorem u i uznajemy, że nie nastąpiło przecięcie.

Kwaterniony – przypomnienie

Kwaternion jest to liczba postaci:

$$q = s + ia + jb + kc,$$

gdzie $s, a, b, c \in \mathbb{R}$ oraz $i^2 = j^2 = k^2 = ijk = -1$

$$ij = k, \qquad jk = i \qquad ki = j,$$

$$ji = -k, \qquad kj = -i, \qquad ik = -j$$

Zbiór kwaternionów oznaczamy przez $\mathbb{H}$.

Kwaternion możemy zapisać w równoważnej postaci:

$$q = [s, \mathrm{v}],$$

gdzie $s \in \mathbb{R}$, $\mathrm{v} \in \mathbb{R}^3$.

Podstawowym działaniem na kwaternionach jakie będziemy potrzebować to mnożenie:

$$[s_a, \mathrm{a}][s_b, \mathrm{b}] = [s_a s_b - \mathrm{a} \cdot \mathrm{b}, s_a \mathrm{b} + s_b \mathrm{a} + \mathrm{a} \times \mathrm{b}]$$

Mnożenie kwaternionów nie jest przemienne!!

Kwaternion sprzężony do kwaternionu $q = [s, \mathrm{v}]$:

$$q^* = [s, -\mathrm{v}]$$

Norma kwaternionu $q = [s, \mathrm{v}]$:

$$|q| = \sqrt{s^2 + \|\mathrm{v}\|^2}.$$

Kwaternion q' nazywamy znormalizowanym, gdy

$$q' = \frac{q}{|q|}$$

Kwaternion odwrotny do q:

$$q^{-1} = \frac{q^*}{|q|^2}$$

Rotację o kąt θ wokół unormowanego wektora $\hat{\mathrm{v}}$ możemy za pomocą kwaternionów wyrazić następująco:

$$qpq^{-1},$$

gdzie

$$q = [\cos\frac{1}{2}\theta, \sin\frac{1}{2}\theta\hat{\mathrm{v}}],$$

orz p to czysty kwaternion przechowujący wektor p, tzn.

$$p = [0, \mathrm{p}].$$

Składanie obrotów sprowadza się do mnożenie kwaternionów, tzn. jeśli kwaterniony q_1 i q_2 reprezentują obroty to kwaternion $q_2 q_1$ reprezentuje złożenie tych obrotów.

Niech $q = [s, \mathrm{v}] = [s, x\mathrm{i} + y\mathrm{j} + z\mathrm{k}]$ i $|q| = 1$ oraz
$p = [0, \mathrm{p}] = [0, x_p\mathrm{i} + y_p\mathrm{j} + z_p\mathrm{k}]$.
Kwaternion qpq^{-1} można zapisać w postaci macierzowej:

$$qpq^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2(y^2+z^2) & 2(xy-sz) & 2(xz+sy) \\ 0 & 2(xy+sz) & 1-2(x^2+z^2) & 2(yz-sx) \\ 0 & 2(xz-sy) & 2(yz+sx) & 1-2(x^2+y^2) \end{bmatrix} \begin{bmatrix} 0 \\ x_p \\ y_p \\ z_p \end{bmatrix}$$

Niech $q = [s, \mathrm{v}] = [s, x\mathrm{i} + y\mathrm{j} + z\mathrm{k}]$ i $|q| = 1$ oraz
$p = [0, \mathrm{p}] = [0, x_p\mathrm{i} + y_p\mathrm{j} + z_p\mathrm{k}]$.
Kwaternion qpq^{-1} można zapisać w postaci macierzowej:

$$qpq^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2(y^2+z^2) & 2(xy-sz) & 2(xz+sy) \\ 0 & 2(xy+sz) & 1-2(x^2+z^2) & 2(yz-sx) \\ 0 & 2(xz-sy) & 2(yz+sx) & 1-2(x^2+y^2) \end{bmatrix} \begin{bmatrix} 0 \\ x_p \\ y_p \\ z_p \end{bmatrix}$$

Macierz

$$\begin{bmatrix} 1-2(y^2+z^2) & 2(xy-sz) & 2(xz+sy) \\ 2(xy+sz) & 1-2(x^2+z^2) & 2(yz-sx) \\ 2(xz-sy) & 2(yz+sx) & 1-2(x^2+y^2) \end{bmatrix}$$

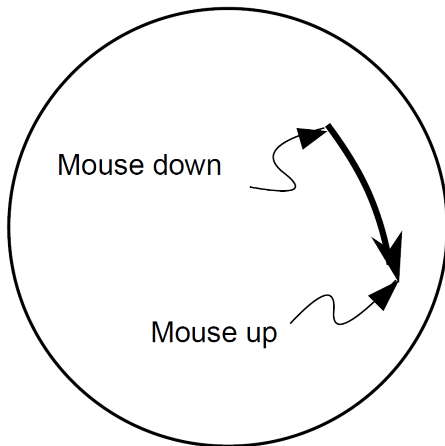
reprezentuje macierz obrotu.

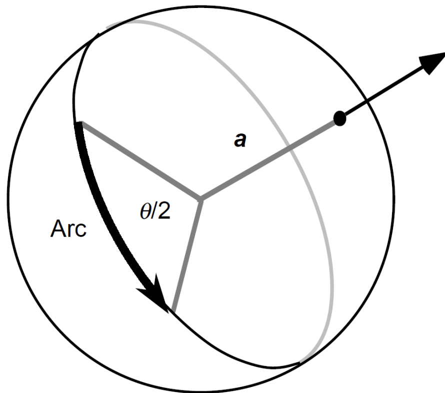
Rotacja sferyczna

Do reprezentacji rotacji sferycznej użyjemy ArcBall przedstawionego przez K.Shoemake'a w 1992 roku.

Rotacja sferyczna

Do reprezentacji rotacji sferycznej użyjemy ArcBall przedstawionego przez K.Shoemake'a w 1992 roku.




a
θ/2
Arc

Nasza sfera będzie miała promień 1 i środek w środku okna programu.
Na początku ustawiamy macierz obrotu na identyczność.
Każdy punkt na ekranie (x, y), który wybierzemy klawiszem myszki lub ciągnąc myszkę z wciśniętym klawiszem będzie najpierw przekształcany do kwadratu $[-1, 1]^2$, oznaczmy taki punkt przez (x_k, y_k). Następnie współrzędna z obliczana jest następująco:

- $r = x_k^2 + y_k^2$,
- jeśli $r > 1$, to
    - $s = 1/\sqrt{r}$
    - $x_k = sx_k$,
    - $y_k = sy_k$,
    - $z_k = 0$,
- w przeciwnym przypadku $z = \sqrt{1 - r}$.

Mając wyznaczony punkt początkowy i końcowy łuku, a tym samym ich wektory p_0, p_1 tworzymy kwaternion reprezentujący obrót:

$$q = [\mathrm{p}_0 \cdot \mathrm{p}_1, \mathrm{p}_0 \times \mathrm{p}_1]$$

Mając wyznaczony punkt początkowy i końcowy łuku, a tym samym ich wektory p_0, p_1 tworzymy kwaternion reprezentujący obrót:

$$q = [\mathsf{p}_0 \cdot \mathsf{p}_1, \mathsf{p}_0 \times \mathsf{p}_1]$$

Następnie kwaternion q przekształcamy w macierz obrotu, którą mnożymy przez bieżącą macierz obrotu.