Zachodniopomorska
Szkoła Biznesu
w Szczecinie

Bartosz Bazyluk

KAMERA W SCENIE 3D

Pojęcie kamery. Implementacja interaktywnej kamery FPP.
Test i bufor głębokości.

Grafika Komputerowa, Informatyka, I Rok

Kamera

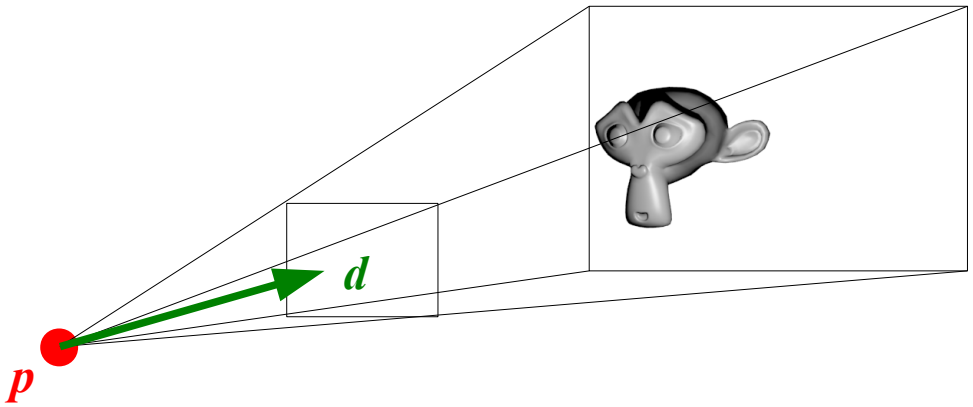
- Jest to wirtualny *koncept* opisujący **sposób oglądania sceny przez obserwatora**, dla którego renderujemy obraz
- Wyróżnia się dwa podstawowe **rodzaje rzutowań**:

Rzut perspektywiczny

Rzut prostokątny
(ang. *orthographic*)

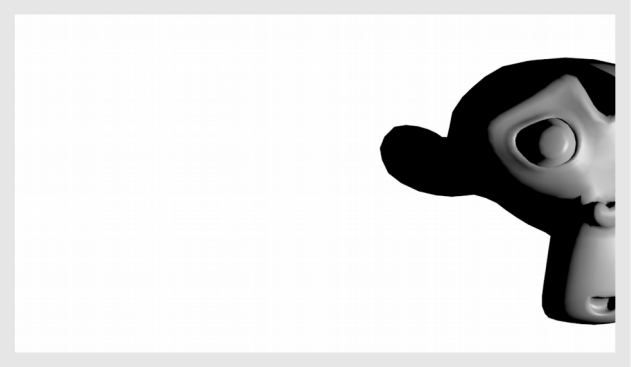
Kamera perspektywiczna

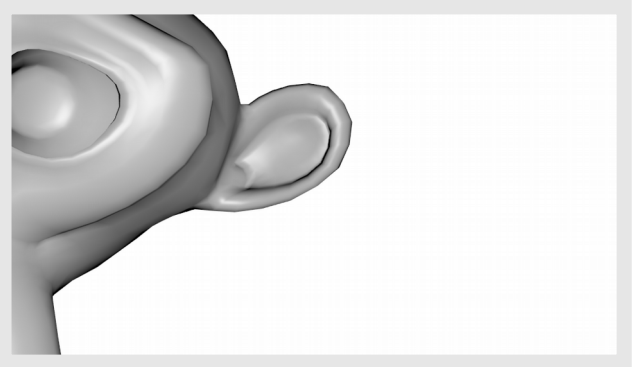
- Kamera musi być **opisana w sposób matematyczny**, aby możliwe było uwzględnienie jej cech w procesie renderowania
- Przede wszystkim musi posiadać zdefiniowane:
  - **Pozycję** (punkt zaczepienia): p
  - **Kierunek widzenia**: d



Kamera perspektywiczna

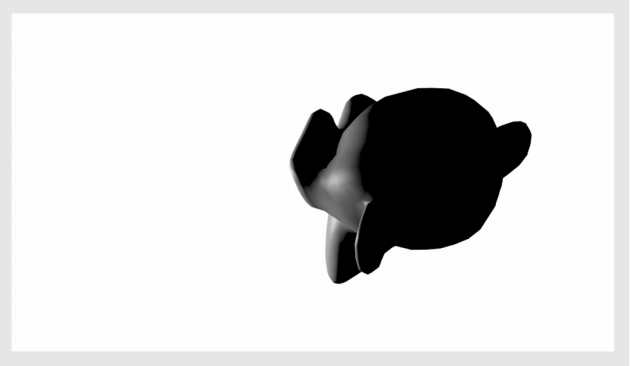
- Położenie w przestrzeni 3D jest po prostu **3-elementowym wektorem**
- Zmiana położenia powoduje **przemieszczenie** naszego wirtualnego "aparatu fotograficznego"





Kamera perspektywiczna

- Kierunek w przestrzeni 3D również jest **3-elementowym wektorem**
  - Kierunek można rozumieć jako wektor leżący na **półprostej** łączącej **punkt zaczepienia** z miejscem, na które kamera ma być **skierowana**
- Kierunek zwyczajowo przechowuje się jako **wektor jednostkowy** ($||d|| = 1$)

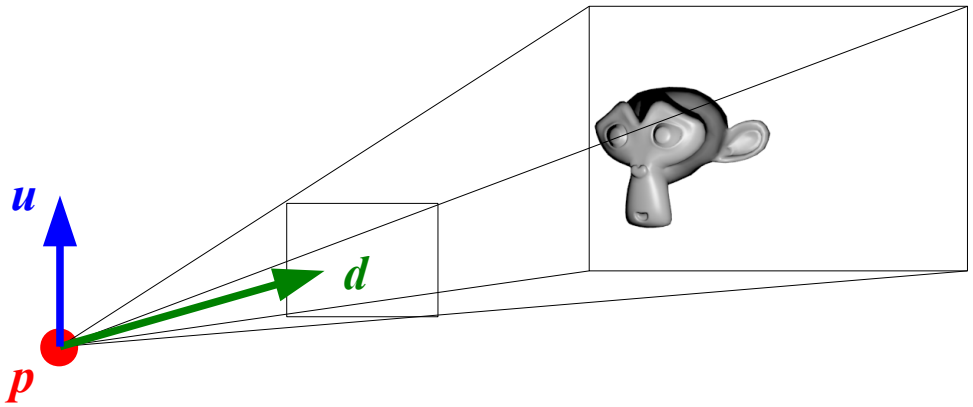


Kamera perspektywiczna

- To jednak **nie wystarczy**, aby **jednoznacznie** określić sposób obserwacji sceny
- Oba poniższe przykłady zostały wyrenderowane **z tego samego miejsca**, **w tym samym kierunku**:

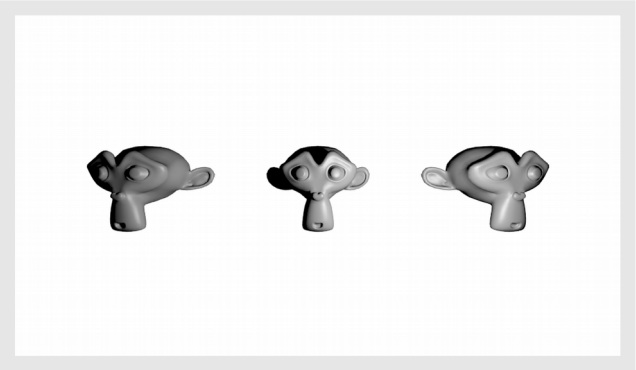
Kamera perspektywiczna

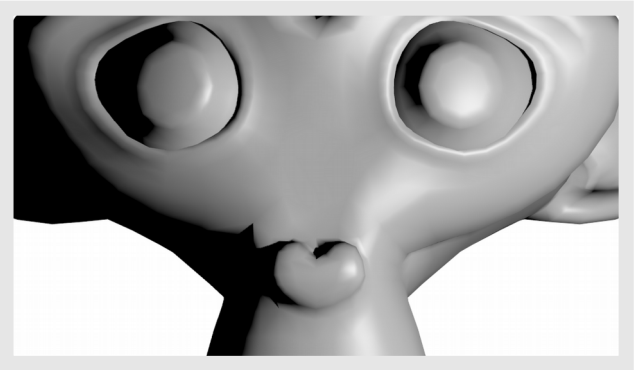
- Konieczne jest wprowadzenie dodatkowo **wektora pionu** (ang. *up vector*): ***u***
- Najczęściej jest to ***(0, 1, 0)*** jeśli za drugą współrzędną przyjmiemy tę skierowaną ku górze świata

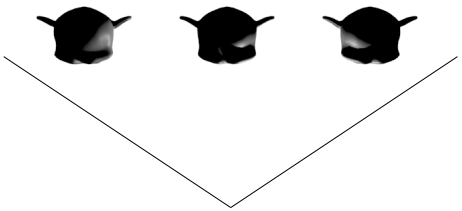


Kamera perspektywiczna

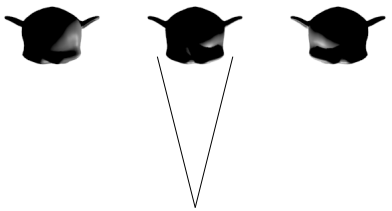
- Dodatkowo kamerę określają:
  - **Kąt widzenia** (najczęściej w stopniach) *fov*
    - Jego zmiana odpowiada **zmianie ogniskowej obiektywu** (popularnie zwanej "*zoomem*" w aparacie fotograficznym)
    - Nie jest tożsamy z przybliżeniem/oddaleniem kamery!







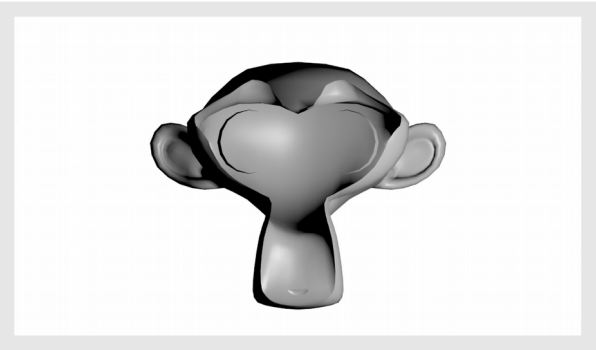
Szeroki kąt
(krótka ogniskowa)

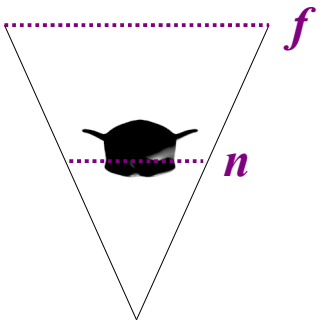


Wąski kąt
(długa ogniskowa)

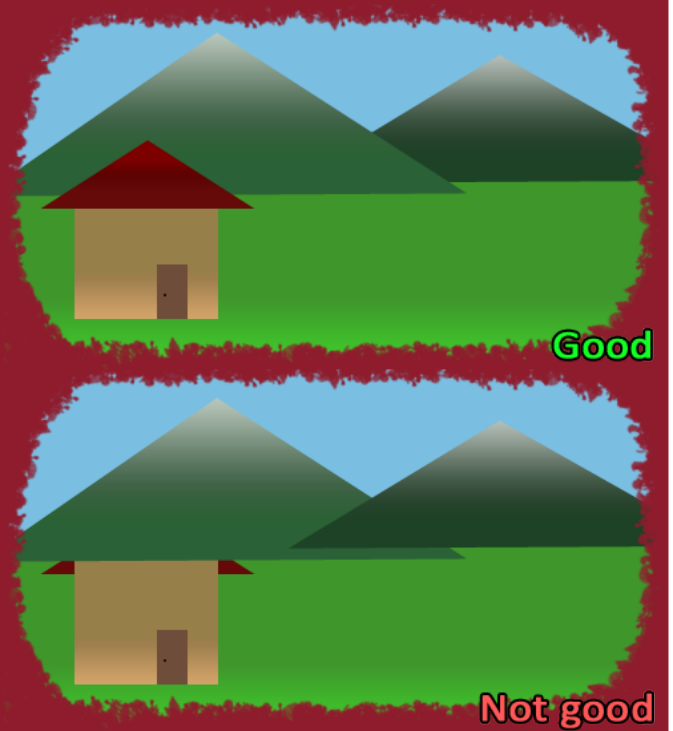
Kamera perspektywiczna

- Dodatkowo kamerę określają:
  - **Odległość płaszczyzn przycinania: bliskiej n i dalekiej f** (ang. *near/far clipping plane*)
    - Określają, od jakiej do jakiej odległości będzie renderowana scena
    - Ma to związek z **precyzją bufora głębokości**
      - Im większy zakres odległości jest renderowany, tym większa szansa powstania problemów typu *z-fighting* – wartości powinny zostać dobrane odpowiednio do charakterystyki danej sceny



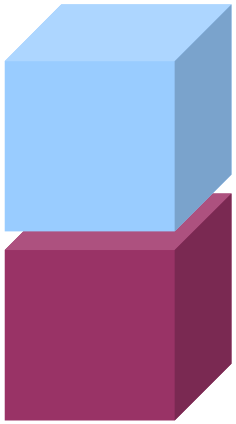


Test głębokości
(ang. *depth test*)

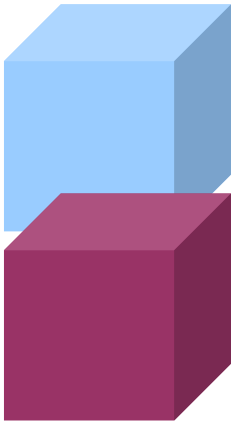


Źródło obrazu:
http://aldream.net

- **Algorytm malarza** (ang. *painter's algorithm*)
  - *Nie jest* dokonywany test głębokości
  - Obiekty przykrywają się **zgodnie z kolejnością rysowania**
  - To, co zostanie narysowane **później**, zawsze przykryje to, co było narysowane **wcześniej**

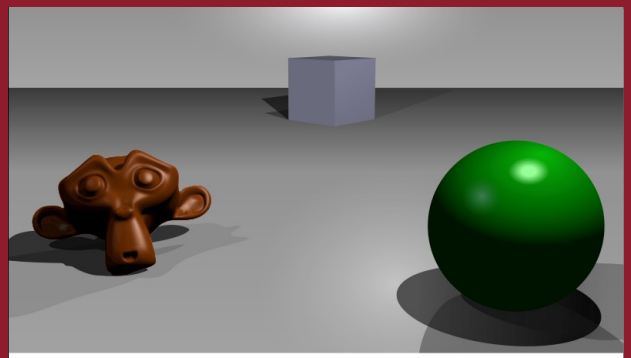


Najpierw narysowano **fioletowy** sześcian

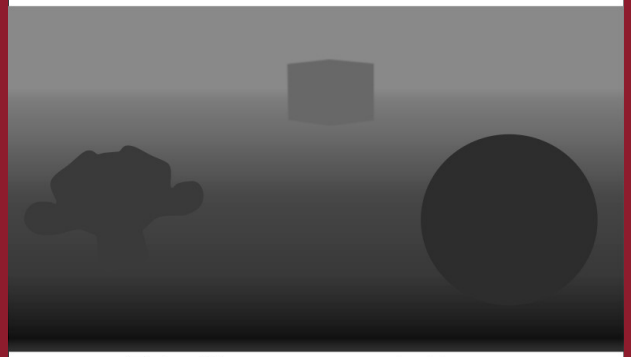


Najpierw narysowano **niebieski** sześcian

Test głębokości
(ang. *depth test*)



A simple three-dimensional scene



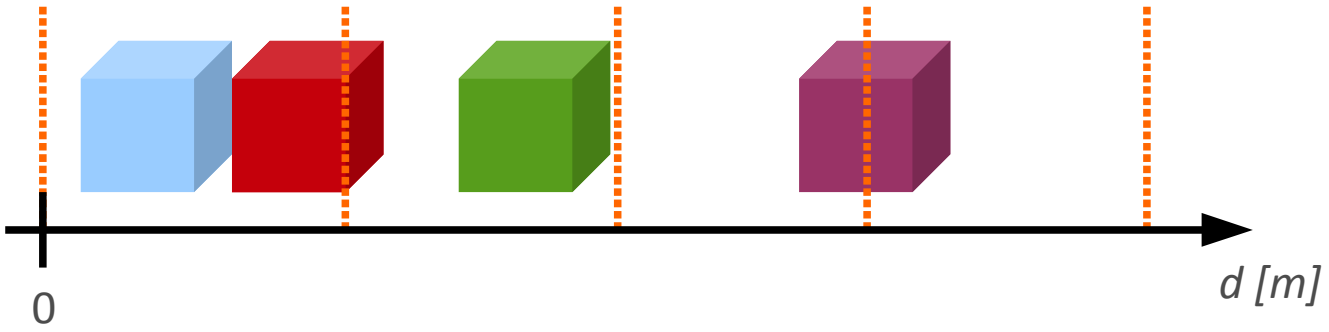
Z-buffer representation

Źródło obrazu:
Wikipedia

- **Algorytm *bufora Z***
  - Oprócz *bufora koloru*, tworzony jest też **bufor głębokości** (*bufor Z*) o odpowiadającym mu rozmiarze
  - W *buforze Z* przechowywane są informacje o **odległości od kamery** tego fragmentu powierzchni obiektu, któremu odpowiada kolor znajdujący się w tym pikselu w buforze koloru
  - Jeśli w wyniku renderowania kolejnego obiektu sceny okaże się, że w danym pikselu bufora już coś się wcześniej znalazło, **wykonywany jest test głębokości**:
    - **Jeśli odległość** od kamery nowego fragmentu powierzchni obiektu **jest mniejsza niż wartość w buforze głębokości**, to nadpisujemy ją nową wartością i nadpisujemy kolor
      - Bo **nowy obiekt jest bliżej kamery**, więc przysłania to, co wcześniej było w buforze koloru
    - **Jeśli odległość** od kamery nowego fragmentu **jest większa niż wartość w buforze głębokości**, to pomijamy nową wartość
      - Bo wcześniej już narysowaliśmy coś, co **przysłania** nasz nowy obiekt

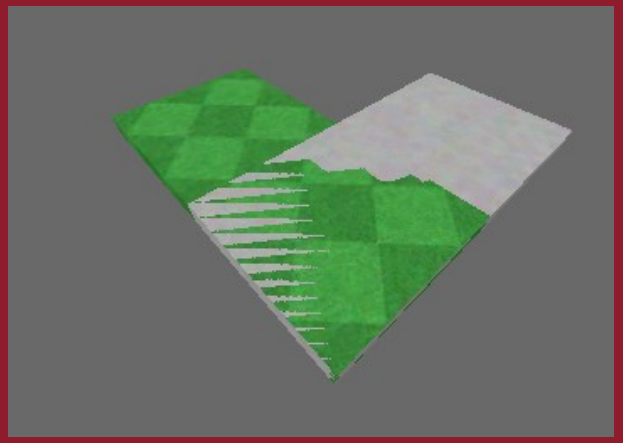
Test głębokości
(ang. *depth test*)

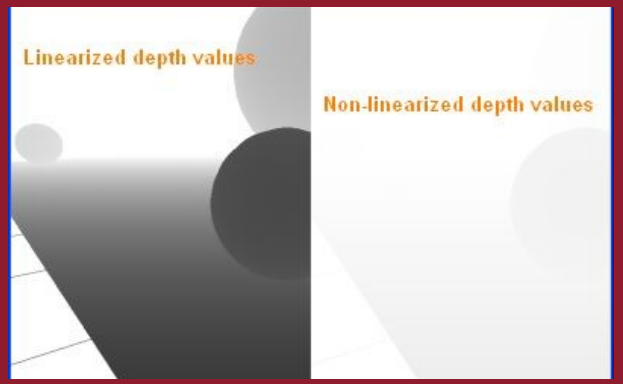
- **Bufor Z**
  - Jest **jednokanałową mapą bitową**, w której każdy na każdy piksel przeznaczono **określoną liczbę bitów**
    - Najczęściej **24 bity** lub **32 bity**
  - W związku z tym możliwe jest zapisanie jedynie **skończonej liczby różnych wartości** odległości od kamery
    - np. wartości całkowite od 0 do 2^{24}-1
  - Prowadzi to do potencjalnych **konfliktów**



Źródło obrazu:
Wikipedia

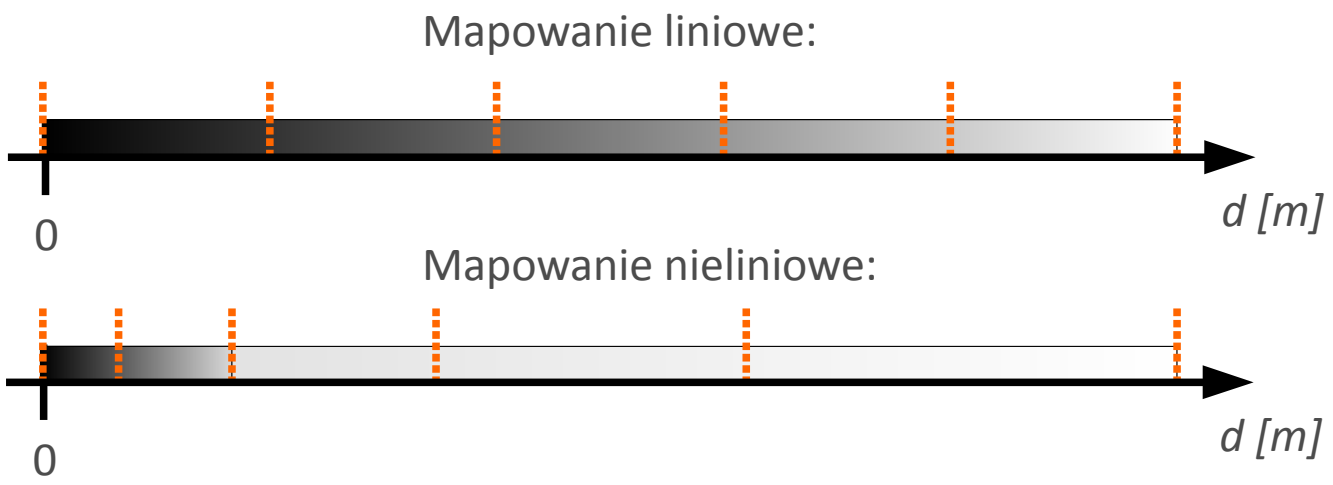
Test głębokości
(ang. *depth test*)



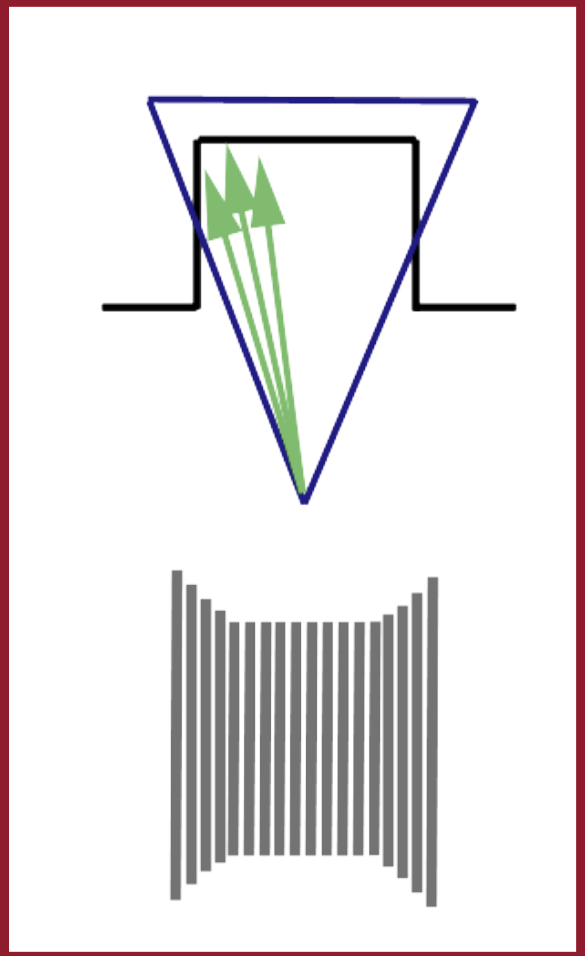


Źródło obrazu:
GeeXLab, JEGX

- **Precyzja** *bufora Z*
  - **Zbyt mała precyzja** prowadzi do **konfliktów**, które objawiają się artefaktem *z-fighting*
  - Należy **ograniczyć zakres odległości**
    - Poprzez użycie **płaszczyzn odcinania**
  - Stosuje się **nieliniowe mapowanie odległości** na wartości *bufora Z*
    - Najczęściej **logarytmiczne**
    - **Większa precyzja dla małych odległości**, mniejsza dla dużych

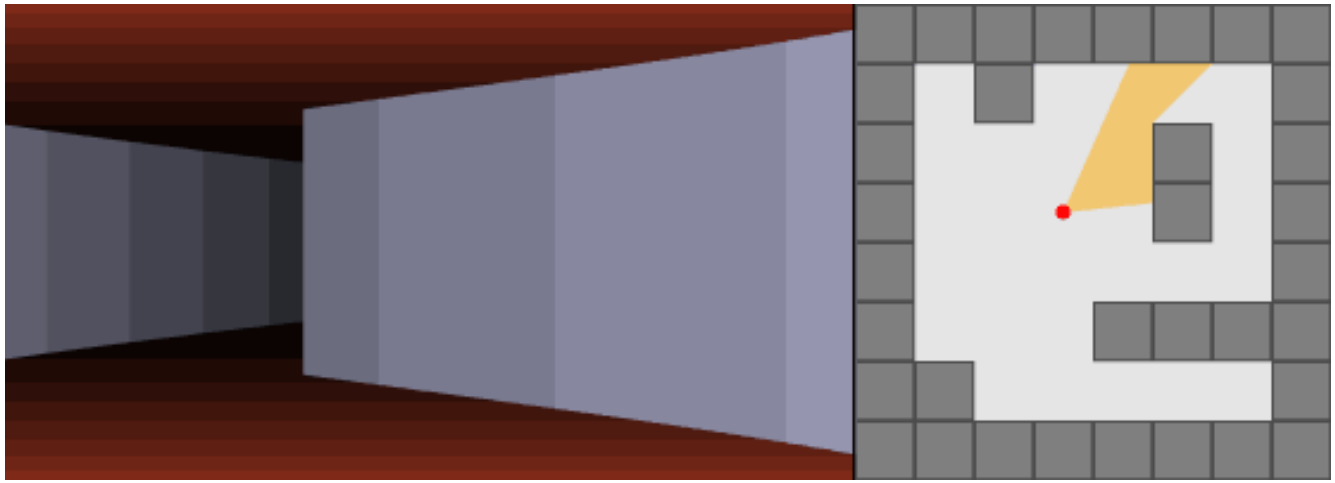


Test widoczności
(ang. *visibility test*)



Źródła obrazów:
Fabien Sanglard, WIkipedia

- Istotny problem w środowiskach 3D: Renderowanie *tylko* tych powierzchni, które **są widoczne**
  - Idea bufora Z przez wiele lat była niemożliwa do użycia w czasie rzeczywistym
- Podejście oparte na **wysyłaniu promieni** (ang. *ray casting*)
  - Jeśli poruszamy się tylko po **płaszczyźnie**, wystarczy tylko tyle promieni, **ile mamy pikseli w poziomie** w buforze klatki
  - **Perspektywa**: na podstawie odległości rysujemy pionową linię o proporcjonalnej wysokości



Test widoczności (ang. *visibility test*)

- Podejście oparte na **wysyłaniu promieni** (ang. *ray casting*)
  - Podejście to, wzbogacone o mapowanie tekstur i proste oświetlenie, zostało użyte w bardzo popularnej w swoich czasach grze ***Wolfenstein 3D***.

Za głównego popularyzatora *ray castingu* jako narzędzia służącego do renderowania ścian pseudo-trójwymiarowej sceny uważa się **Johna Carmacka**, założyciela *id Software*. Pierwsze użycie tego podejścia miało miejsce w grze Hovertank 3D.

Źródła obrazów:
id Software, Wikipedia

Test głębokości
(ang. *depth test*)

- Można stosować **komponowanie klatki** za pomocą zarówno *testu głębokości*, jak i *algorytmu malarza*.
  - **Interfejs użytkownika** zwykle nanosimy na wyrenderowaną wizualizację świata *bez* użycia testu głębokości
  - Kiedyś stosowano uproszczenia w celu przyspieszenia renderowania, np. samochód gracza mógł być rysowany na tle otoczenia, bo rzadko kiedy dochodzi do sytuacji w której miałby być czymś przysłonięty

Źródło obrazu:
Virgin Interactive

Kamera perspektywiczna

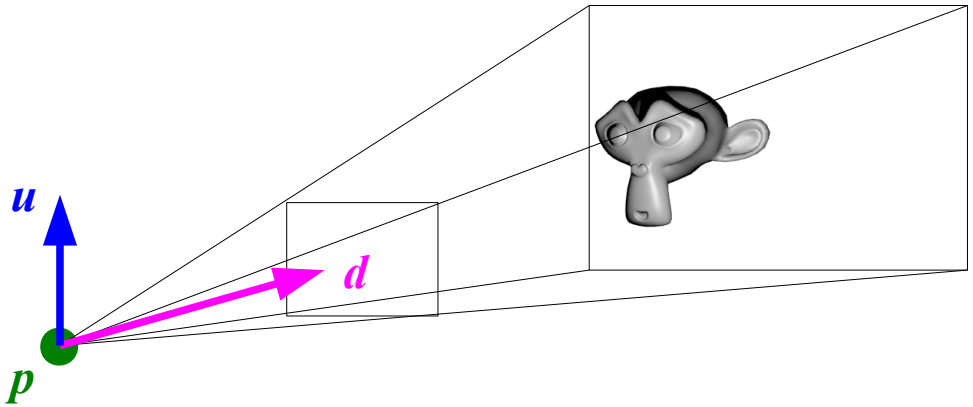
- Dodatkowo kamerę określają:
  - **Proporcje** boków podstawy ściętego ostrosłupa (ang. *frustum*)
    - Stosunek szerokości do wysokości
      - 4:3, 16:9, 16:10, …

Kamera perspektywiczna

- Podsumowując, do **jednoznacznego opisania kamery** potrzebne są wartości następujących atrybutów:
  - Cechy **widoku**:
    - Położenie
    - Kierunek
    - Wektor pionu
  - Cechy **projekcji**:
    - Kąt widzenia
    - Odległości płaszczyzn przycinania
    - Proporcje

Kamera w OpenGL

- Aby opisać jednoznacznie położenie kamery, należy określić:
  - Pozycję ***p***
  - Kierunek widzenia ***d***
  - Wektor pionu ***u***
- Każdy z tych elementów w przestrzeni 3D jest 3-elementowym wektorem

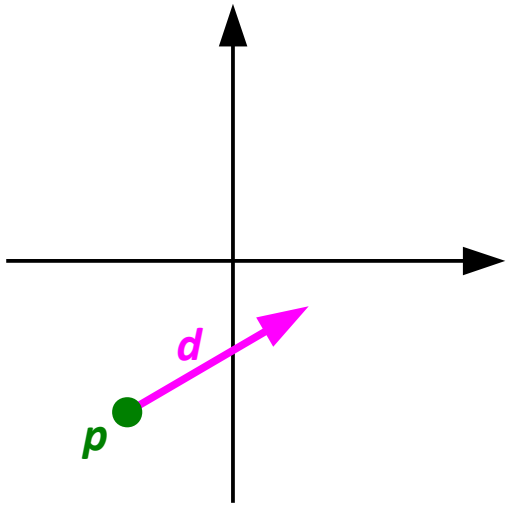


Kamera w OpenGL

- Aby łatwo uzyskać w OpenGL **macierz transformacji** odpowiadającą kamerze **umieszczonej w zadanym punkcie**, patrzącą **w zadanym kierunku** i o **zadanym wektorze pionu**, najlepiej posłużyć się funkcją:
  - **gluLookAt**(
    eyeX, eyeY, eyeZ,
    lookatX, lookatY, lookatZ,
    upX, upY, upZ
)
    - ***eye**** - współrzędne *XYZ* punktu zaczepienia kamery
    - ***lookat**** - współrzędne *XYZ* punktu, na który ma spoglądać kamera
    - ***up**** - współrzędne *XYZ* wektora pionu
      - W dalszych rozważaniach dla uproszczenia przyjmiemy, że wektor pionu zawsze będzie równy *(0;1;0)*

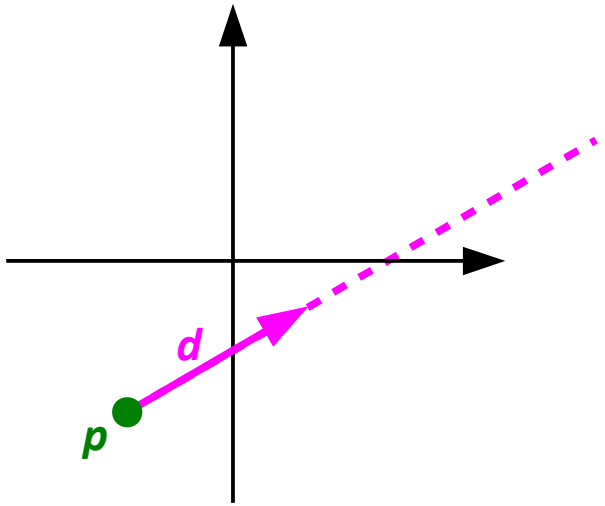
Kamera w OpenGL

- Należy zwrócić uwagę, że *gluLookAt()* nie przyjmuje kierunku widzenia kamery, a punkt na który kamera ma spoglądać
  - Jest to szczególnie istotne, gdy naszym zadaniem jest zaprogramowanie **interaktywnej kamery pierwszoosobowej**
    - W takim przypadku dużo wygodniej jest przechowywać kierunek widzenia jako **wektor jednostkowy**
    - Na podstawie **pozycji** kamery o raz **kierunku** można **łatwo uzyskać współrzędne punktu**, który można przekazać do *gluLookAt()*.



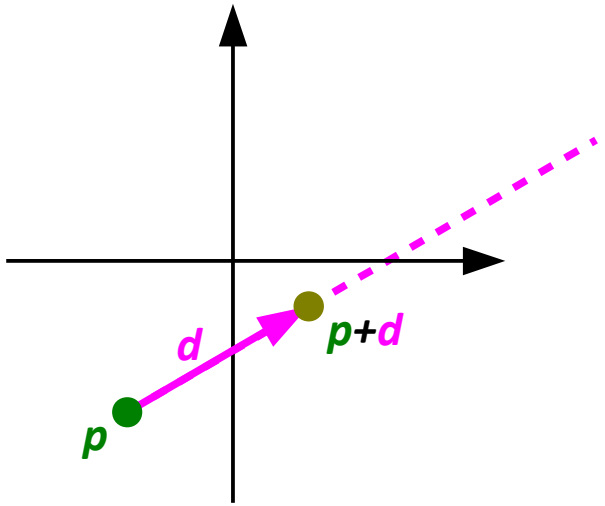
Kamera w OpenGL

- Należy zwrócić uwagę, że *gluLookAt()* nie przyjmuje kierunku widzenia kamery, a punkt na który kamera ma spoglądać
  - Jest to szczególnie istotne, gdy naszym zadaniem jest zaprogramowanie **interaktywnej kamery pierwszoosobowej**
    - W takim przypadku dużo wygodniej jest przechowywać kierunek widzenia jako **wektor jednostkowy**
    - Na podstawie **pozycji** kamery o raz **kierunku** można **łatwo uzyskać współrzędne punktu**, który można przekazać do *gluLookAt()*.



Kamera w OpenGL

- Należy zwrócić uwagę, że *gluLookAt()* nie przyjmuje kierunku widzenia kamery, a punkt na który kamera ma spoglądać
  - Jest to szczególnie istotne, gdy naszym zadaniem jest zaprogramowanie **interaktywnej kamery pierwszoosobowej**
    - W takim przypadku dużo wygodniej jest przechowywać kierunek widzenia jako **wektor jednostkowy**
    - Na podstawie **pozycji** kamery o raz **kierunku** można **łatwo uzyskać współrzędne punktu**, który można przekazać do *gluLookAt()*.



Punkt o współrzędnych ***p+d*** bez wątpienia znajduje się na osi widzenia kamery znajdującej się w punkcie *p*, skoro wektor *d* wskazuje jej kierunek widzenia.

Stan kamery

Strukturę/klasę przechowującą dane wektora wygodnie jest **rozbudować** o przydatne metody (np. długość, normalizacja, iloczyn wektorowy) i **przeciążone operatory** dla częstych operacji arytmetycznych.

Podejść do implementacji kamery jest wiele. Przytoczone tutaj jest wygodne dla prostych scen i stosunkowo łatwe w zrozumieniu.

Poszukując rozwiązania bardziej uniwersalnego, warto przyjrzeć się podejściu opartemu na kwaternionach.

- Reprezentacja stanu kamery w pamięci
  - Wygodne jest zapamiętanie **trzech wektorów 3-elementowych**:
    - **Położenia**
    - **Kierunku patrzenia**
    - **Pionu**
  - Sugerowane jest utworzenie **struktury** przechowującej trzy składowe *XYZ*:

```
struct vec3 {
        float x, y, z;
};
```

  - Wówczas **stan kamery** można przedstawić następująco:

```
struct SCameraState {
        vec3 pos;
        vec3 dir;
        vec3 up;
};
```

Stan kamery

- Jako że dla uproszczenia przyjęliśmy, że **wektor pionu jest stały**, możemy go pominąć
- Przydatne może okazać się przechowanie **aktualnej prędkości** przesuwania kamery
  - Np. w celu płynnego wygaszania ruchu
- Zatem struktura stanu kamery może przyjąć następującą postać:

```
struct SCameraState {
	vec3 pos;
	vec3 dir;
	float speed;
};
```

- Na potrzeby zadania utworzymy globalną instancję tej struktury i nazwiemy ją *player*

```
SCameraState player;
```

Stan kamery

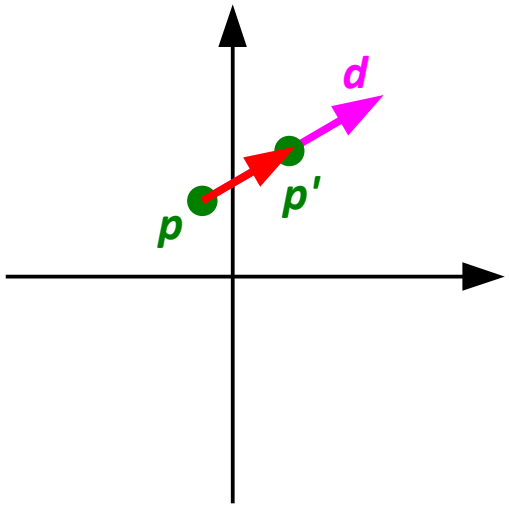
- W takiej sytuacji, możemy każdorazowo zasilać wywołanie *gluLookAt()* następującymi danymi:

```
gluLookAt(
        player.pos.x,
        player.pos.y,
        player.pos.z,
        player.pos.x + player.dir.x,
        player.pos.y + player.dir.y,
        player.pos.z + player.dir.z,
        0,
        1,
        0
);
```

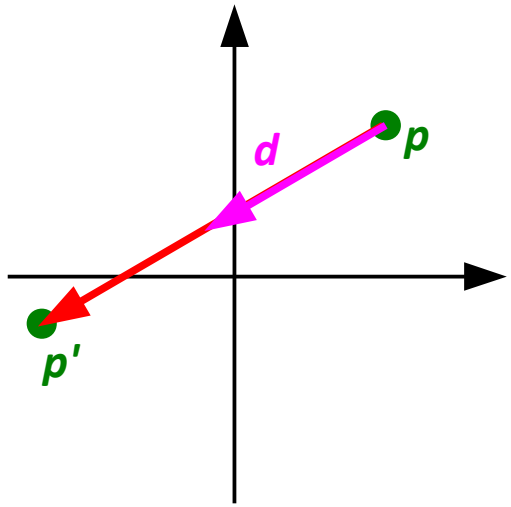
Ruch kamery w przód/tył

- Aby zrealizować taki ruch, trzeba zmodyfikować położenie kamery
  - kierunek i pion pozostaną bez zmian
- "Przód" w danym momencie odpowiada wektorowi kierunku kamery
- Pozycję po przesunięciu obliczamy w prosty sposób:

$$p' = p + speed \cdot d$$



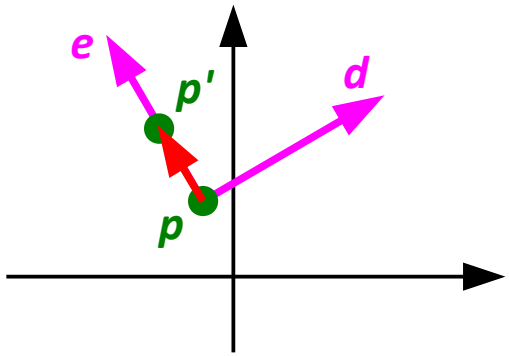
Mała prędkość (mnożnik)



Duża prędkość (mnożnik)

Ruch kamery w bok

- Aby uwzględnić kierunek widzenia kamery, ruszając się na boki należy poruszać się po **kierunku prostopadłym**
  - W założeniu mamy poruszać się tylko **po płaszczyźnie *XZ***



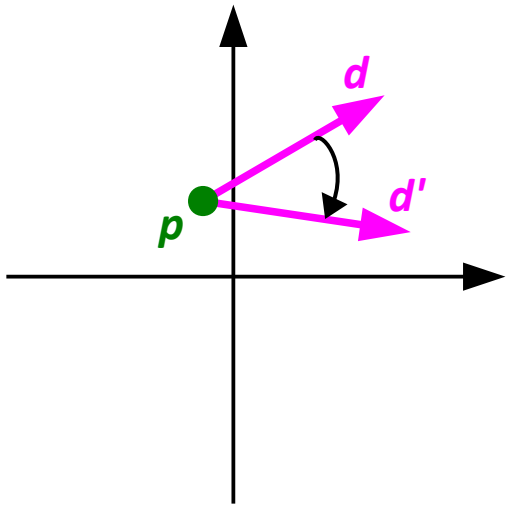
$$p' = p + speed \cdot e, \quad e \perp d$$

- Jak wyznaczyć **wektor prostopadły do danego**?

$$d = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \Rightarrow e = \begin{pmatrix} -d_z \\ 0 \\ d_x \end{pmatrix}$$

Obrót kamery

- Obrót kamery jest zmianą kierunku jej widzenia, bez wpływu na punkt zaczepienia
  - W założeniu mamy poruszać się tylko **po płaszczyźnie *XZ***
  - Rezultatem jest wektor kierunku **obrócony o zadany kąt**
    - **Jak wyliczyć jego współrzędne?**

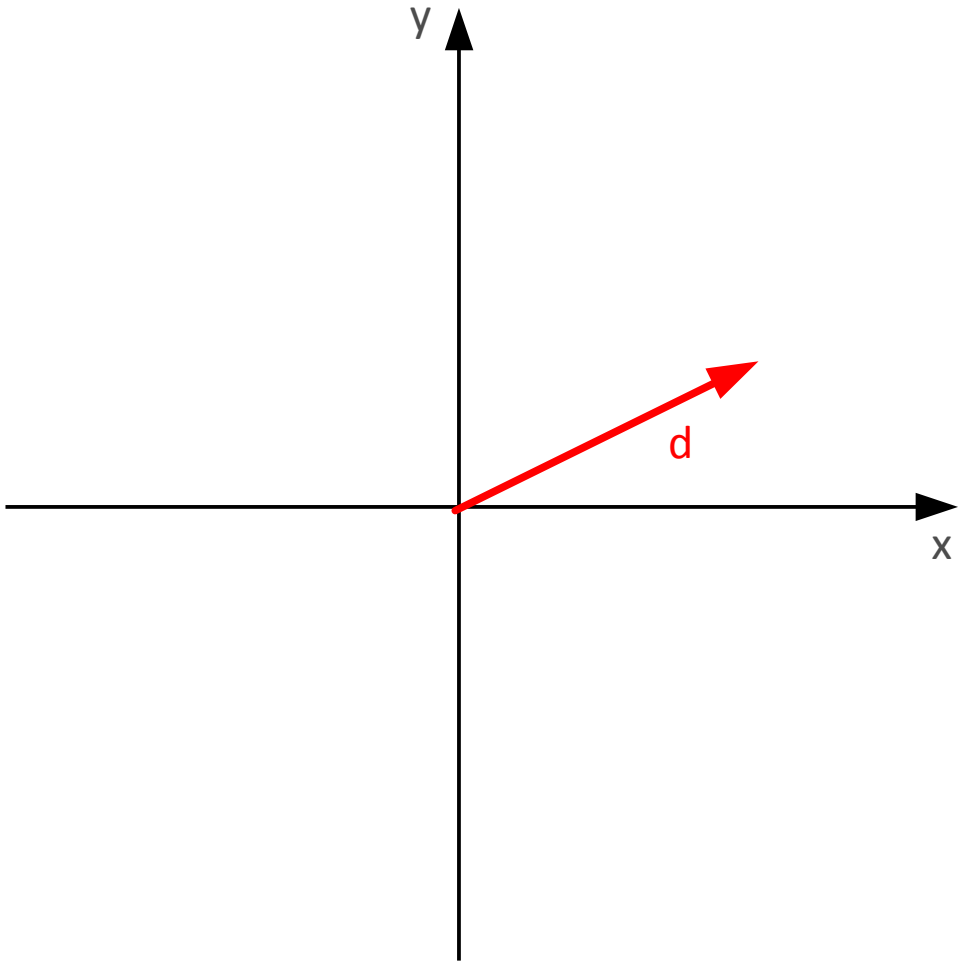


Obrót wektora kierunku w 2D

Dla uproszczenia sprowadźmy problem do **dwóch wymiarów**.

Będziemy dokonywać obrotu tylko **w obrębie jednej płaszczyzny**, którą roboczo nazwiemy *x/y*.

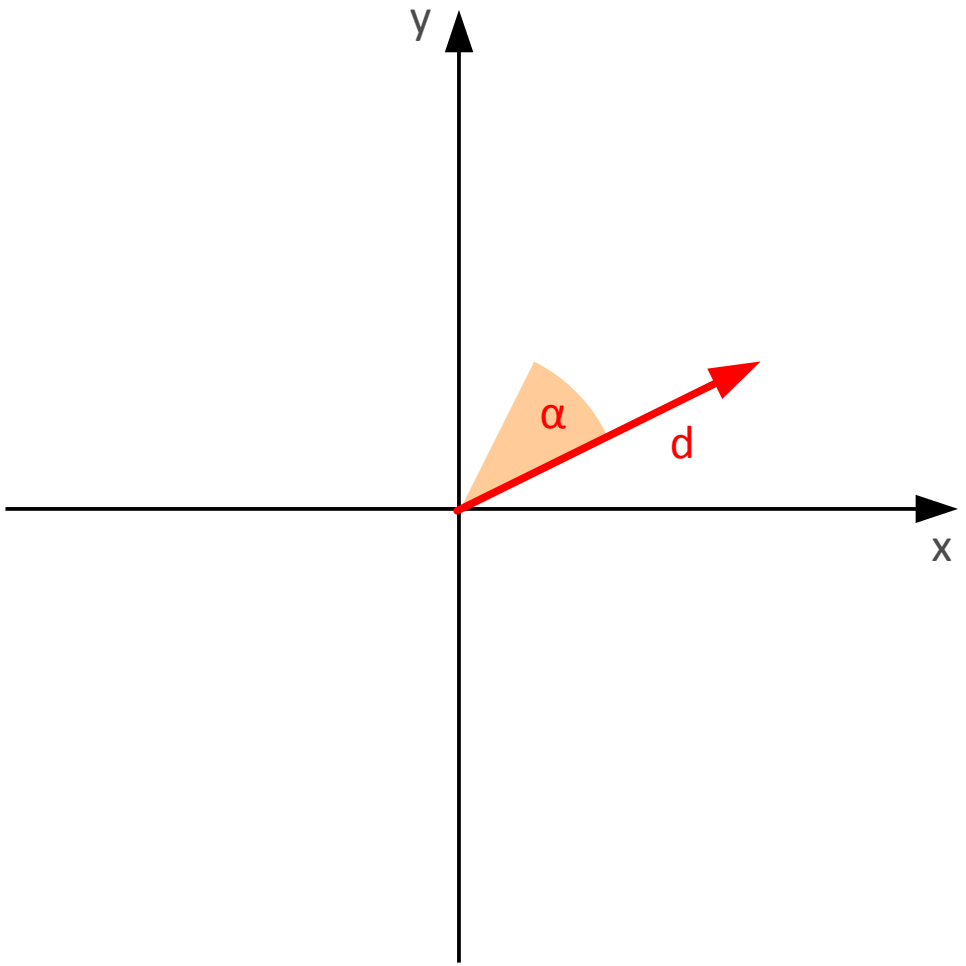
Chcąc poruszać się po płaszczyźnie *x/z* układu współrzędnych *OpenGL* należy odpowiednio zmienić oznaczenie osi!



Obrót wektora kierunku w 2D

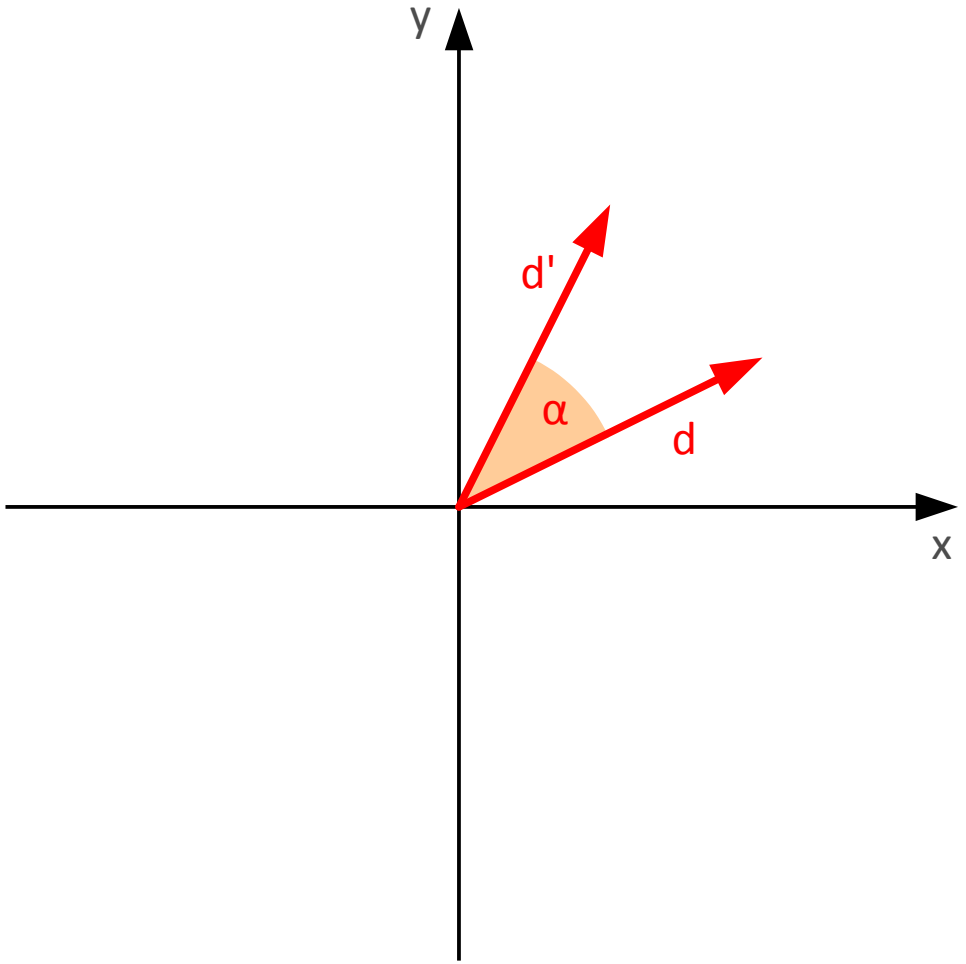
Chcemy dokonać **obrotu** kamery o **zadany kąt *α***.

Miara tego kąta może być rozumiana jako „krok” wykonany podczas tej jednej operacji obracania.



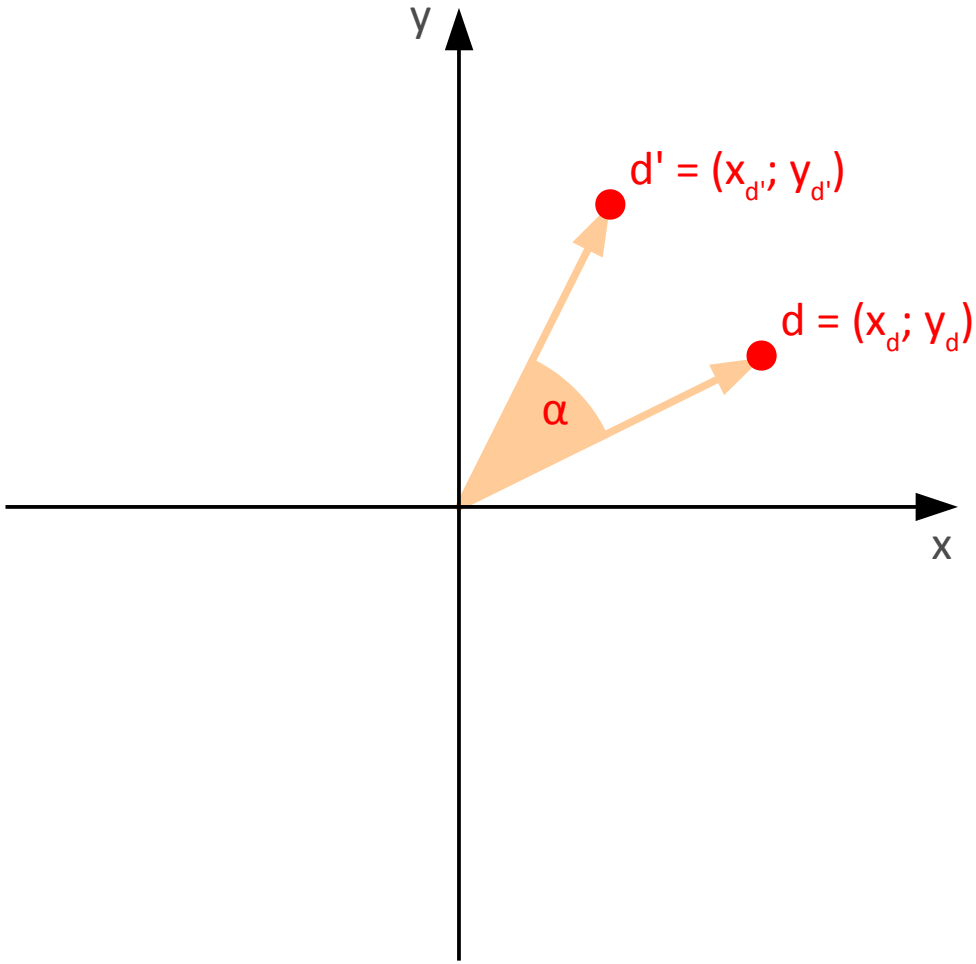
Obrót wektora kierunku w 2D

Naszym wynikiem powinien być **nowy wektor kierunku *d'***, uzyskany po obróceniu pierwotnego wektora *d* wokół początku układu współrzędnych.



Obrót wektora kierunku w 2D

Można przyjąć, że wektory zaczepione w początku układu współrzędnych to **położenia punktów** znajdujących się na ich końcach.



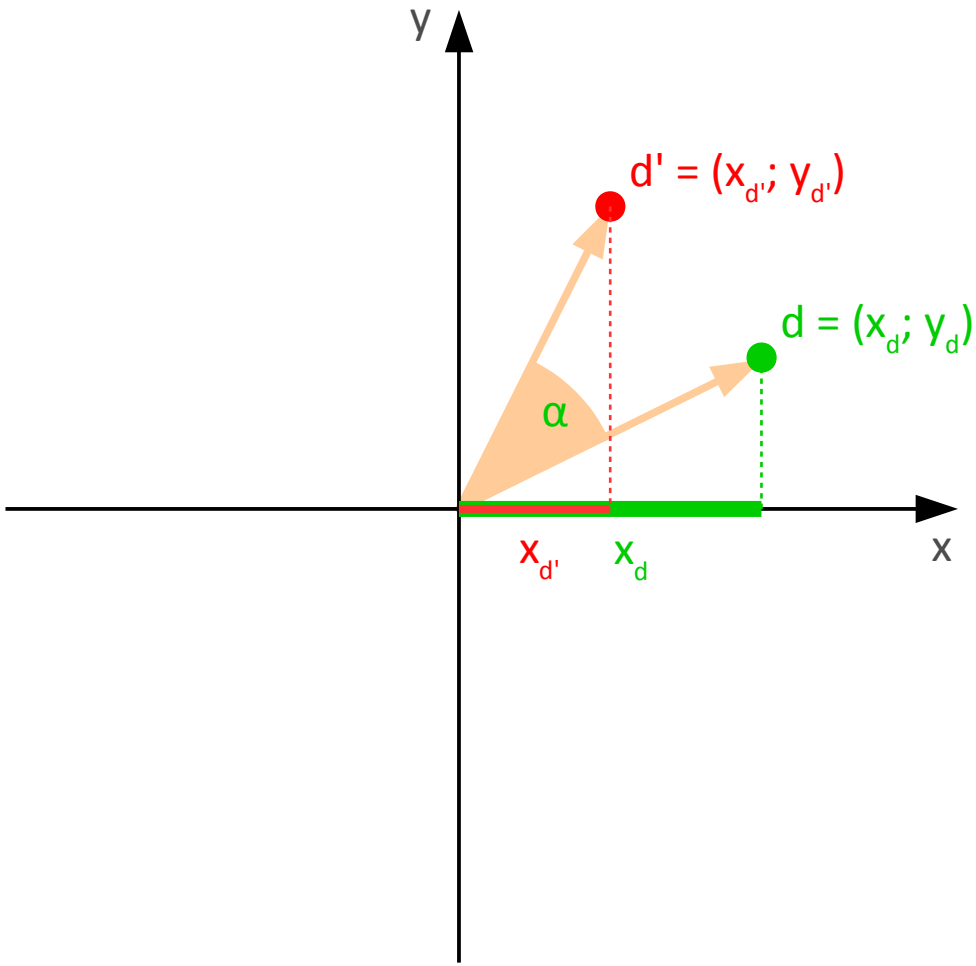
Obrót wektora kierunku w 2D

Rozpatrzmy najpierw samą współrzędną *x*.

Znamy x_d

Znamy miarę kąta α

Nie znamy natomiast $x_{d'}$



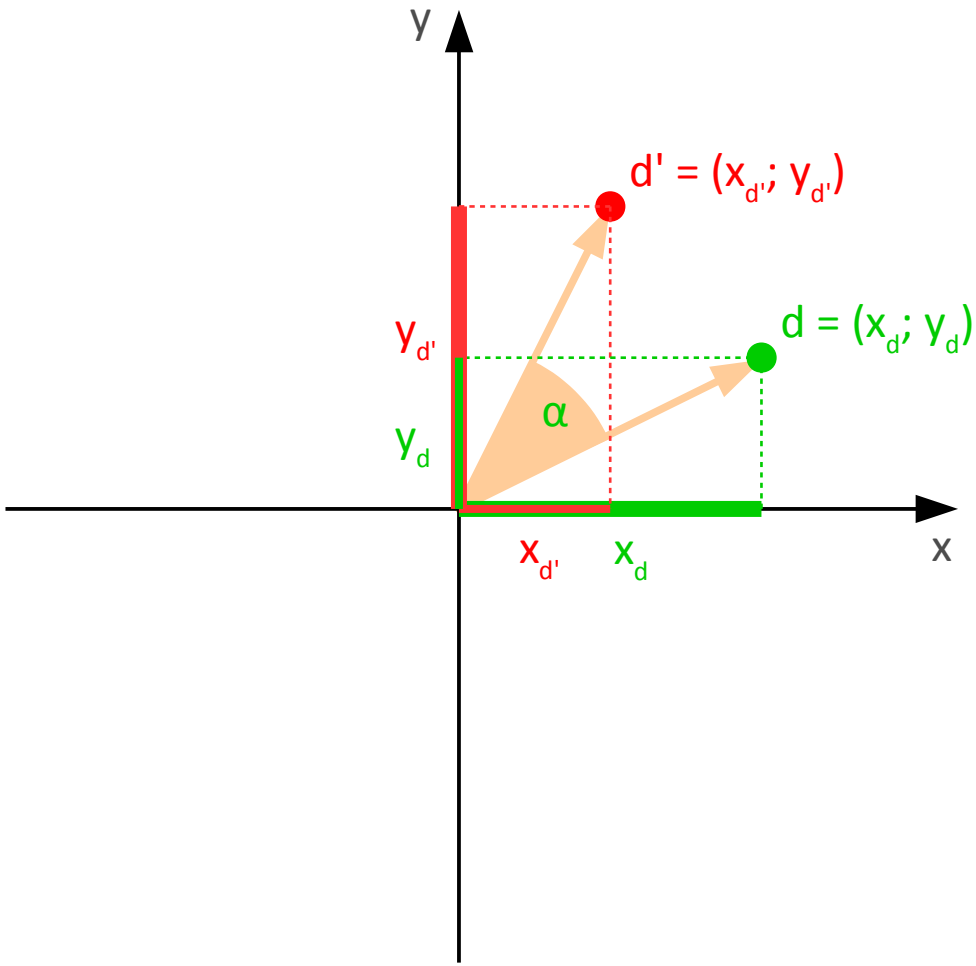
Obrót wektora kierunku w 2D

Analogiczna sytuacja zachodzi dla współrzędnej *y*.

Znamy y_d

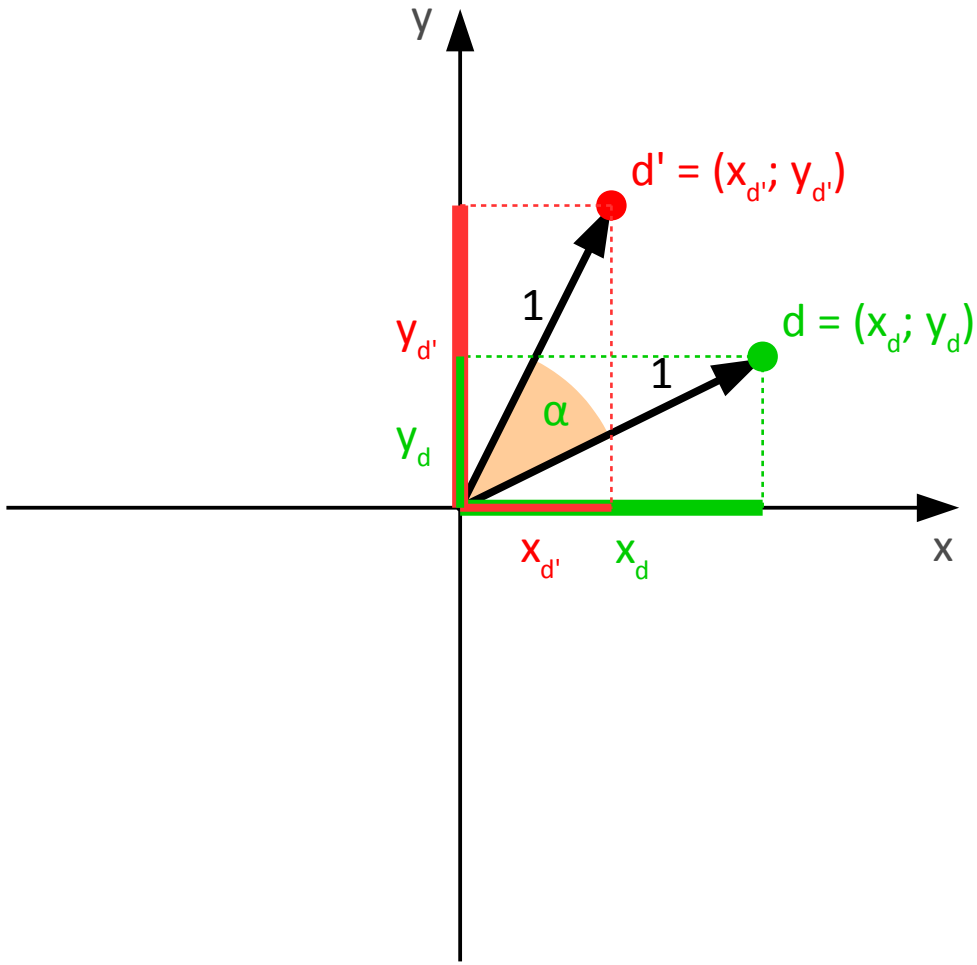
Znamy miarę kąta α

Nie znamy natomiast $y_{d'}$



Obrót wektora kierunku w 2D

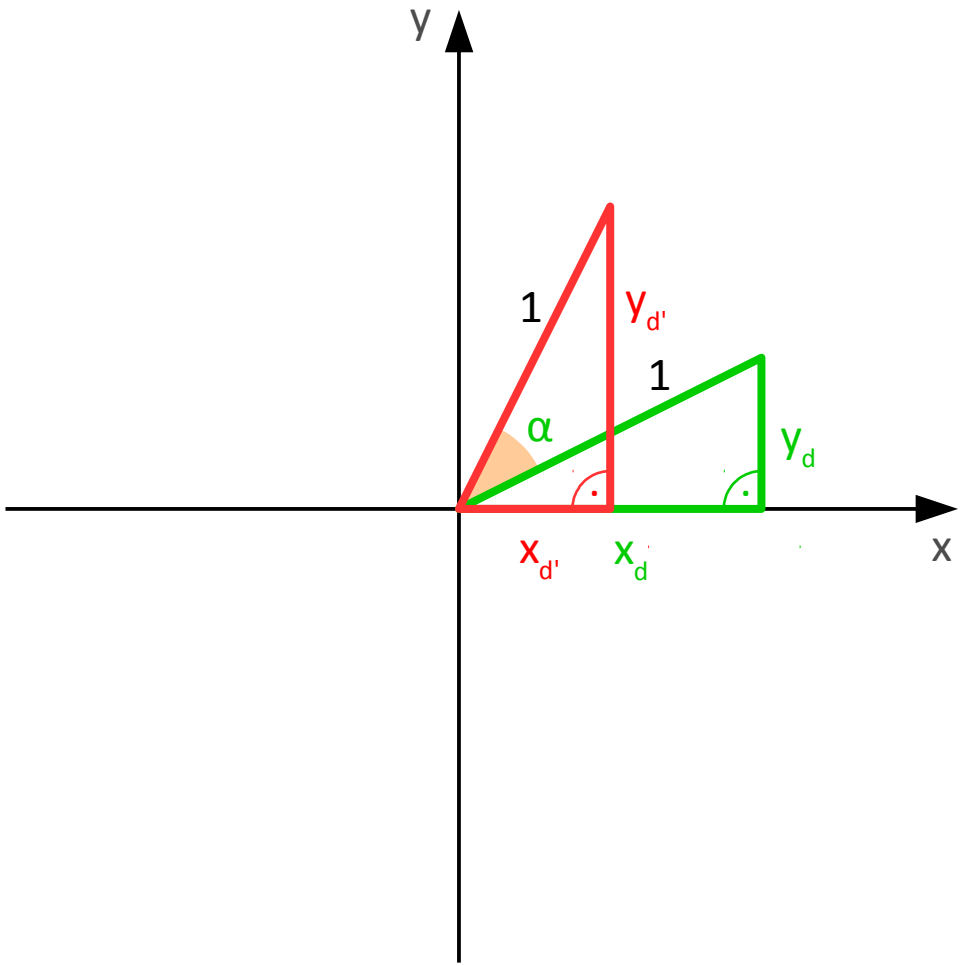
Dodatkowo wiemy, że **długości wektorów *d* i *d'* są równe 1** – ponieważ przyjęliśmy takie założenie odnośnie wektora *d*, a także chcemy aby nasz *d'* zachował tę własność.



Obrót wektora kierunku w 2D

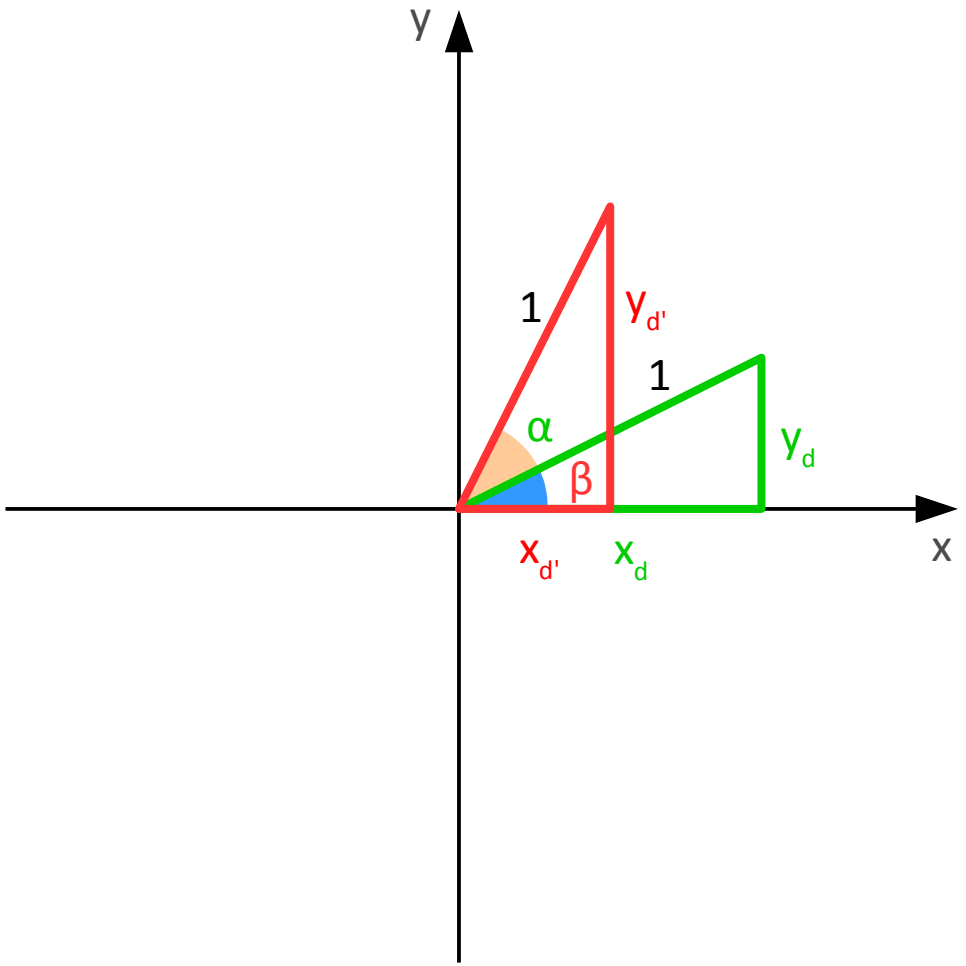
Warto zwrócić uwagę, że mamy tu do czynienia z dwoma **trójkątami prostokątnymi**.

$\{ x_d , x_{d'} , y_d , y_{d'} \}$ są długościami kolejnych **przyprostokątnych**, zaś **przeciwprostokątne** mają długość 1.



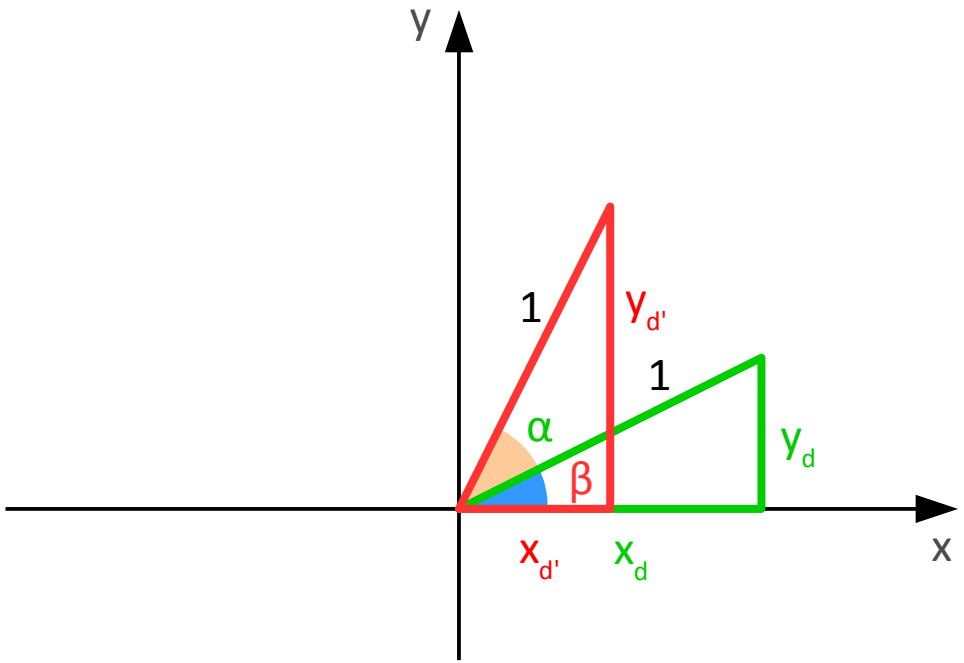
Obrót wektora kierunku w 2D

Gdyby znać miarę kąta **α+β**, znając długość przeciwprostokątnej (jest równa 1), możliwe byłoby wyznaczenie długości przyprostokątnej *czerwonego* trójkąta (czyli $x_{d'}$ lub $y_{d'}$) z pomocą odpowiednich funkcji trygonometrycznych.



Obrót wektora kierunku w 2D

Miarę kąta β możemy znaleźć posługując się znanymi długościami boków *zielonego* trójkąta.



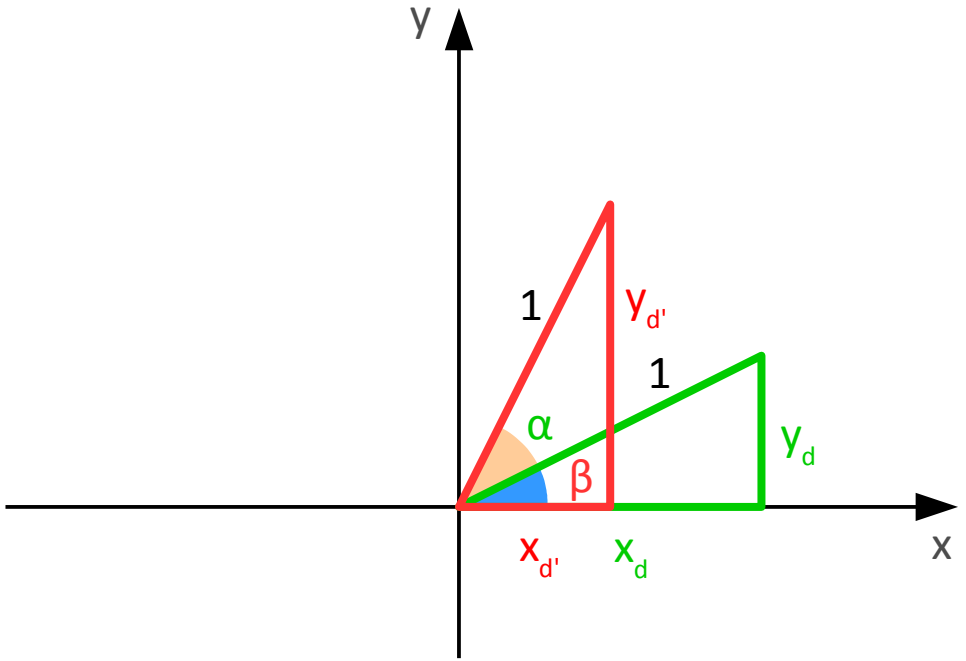
$$\tan(\beta) = \frac{y_d}{x_d}$$

Obrót wektora kierunku w 2D

Najwygodniej, z uwagi na możliwość wystąpienia kąta w dowolnej ćwiartce układu współrzędnych, posłużyć się funkcją *atan2*.

Funkcja cyklometryczna *atan2* jest dwuargumentową odmianą funkcji *arcus tangens*, biorącą pod uwagę ćwiartkę układu w której znajduje się zadany kąt.

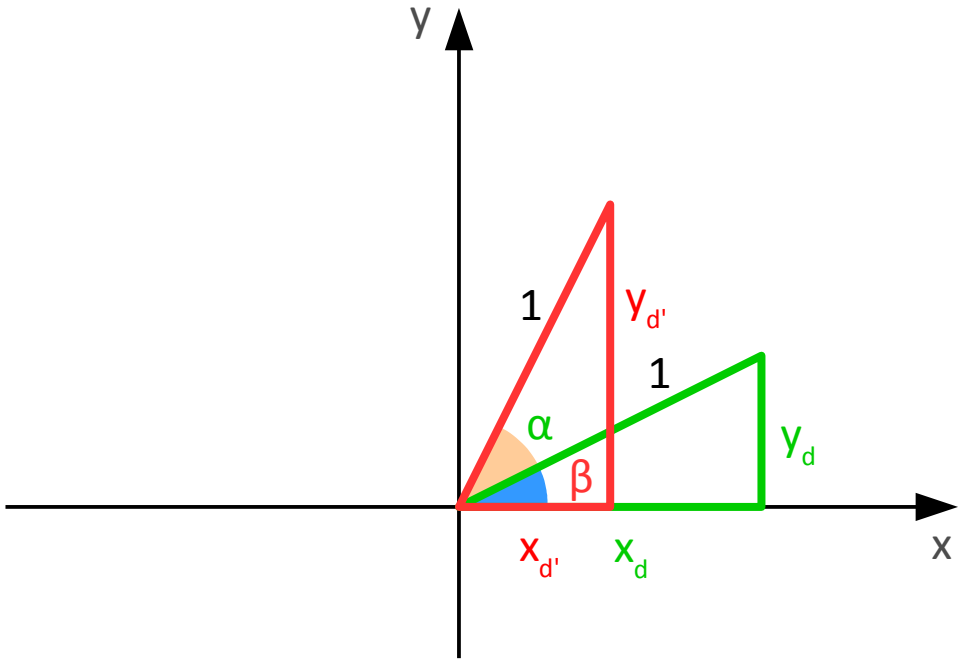
[http://en.wikipedia.org/wiki/Atan2]



$$\tan(\beta)=\frac{y_d}{x_d} \quad \Rightarrow \quad \beta=atan\,2(y_d, x_d)$$

Obrót wektora kierunku w 2D

Kiedy już znamy miarę kąta β, możemy przystąpić do obliczania długości przyprostokątnych *czerwonego* trójkąta korzystając z funkcji trygonometrycznych *sinus* i *cosinus*.



$$\tan(\beta)=\frac{y_d}{x_d} \quad \Rightarrow \quad \beta=atan\,2(y_d, x_d)$$

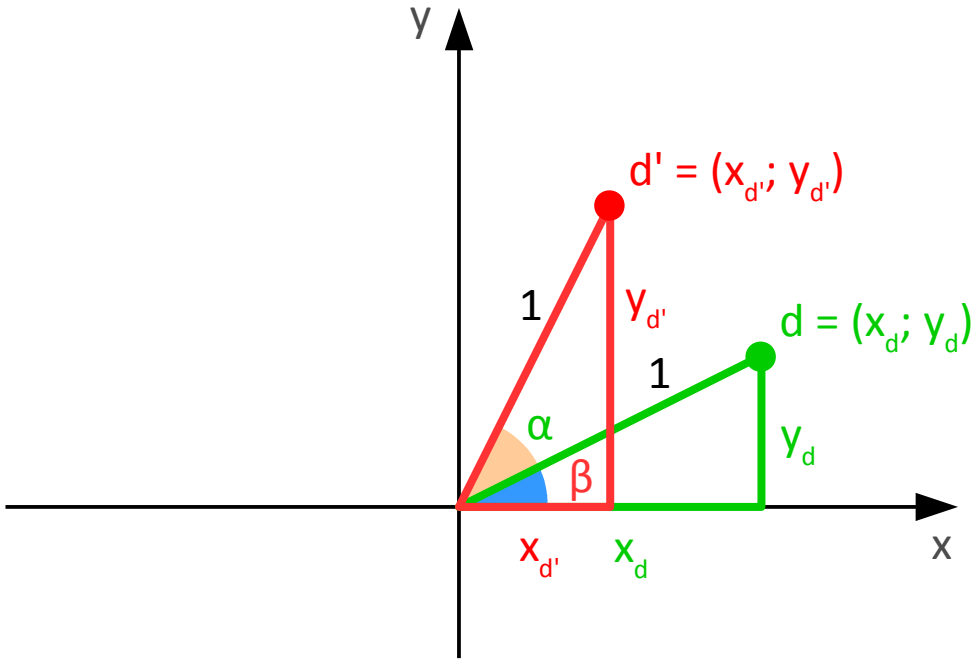
$$\frac{x_{d'}}{|d'|}=\cos(\beta+\alpha)$$

$$\frac{y_{d'}}{|d'|}=\sin(\beta+\alpha)$$

Obrót wektora kierunku w 2D

Pamiętamy, że długość przeciwprostokątnych jest równa 1, co dodatkowo upraszcza obliczenia.

W ten sposób otrzymujemy ostateczne współrzędne punktu *d'* po obrocie punktu *d* o kąt *α* wokół początku układu współrzędnych.



$$\tan(\beta)=\frac{y_d}{x_d} \quad \Rightarrow \quad \beta=atan\,2(y_d, x_d)$$

$$x_{d'}=\cos(\beta+\alpha)$$

$$y_{d'}=\sin(\beta+\alpha)$$

Zachodniopomorska
Szkoła Biznesu
w Szczecinie

Bartosz Bazyluk

KAMERA W SCENIE 3D

Pojęcie kamery. Implementacja interaktywnej kamery FPP.
Test i bufor głębokości.

Grafika Komputerowa, Informatyka, I Rok