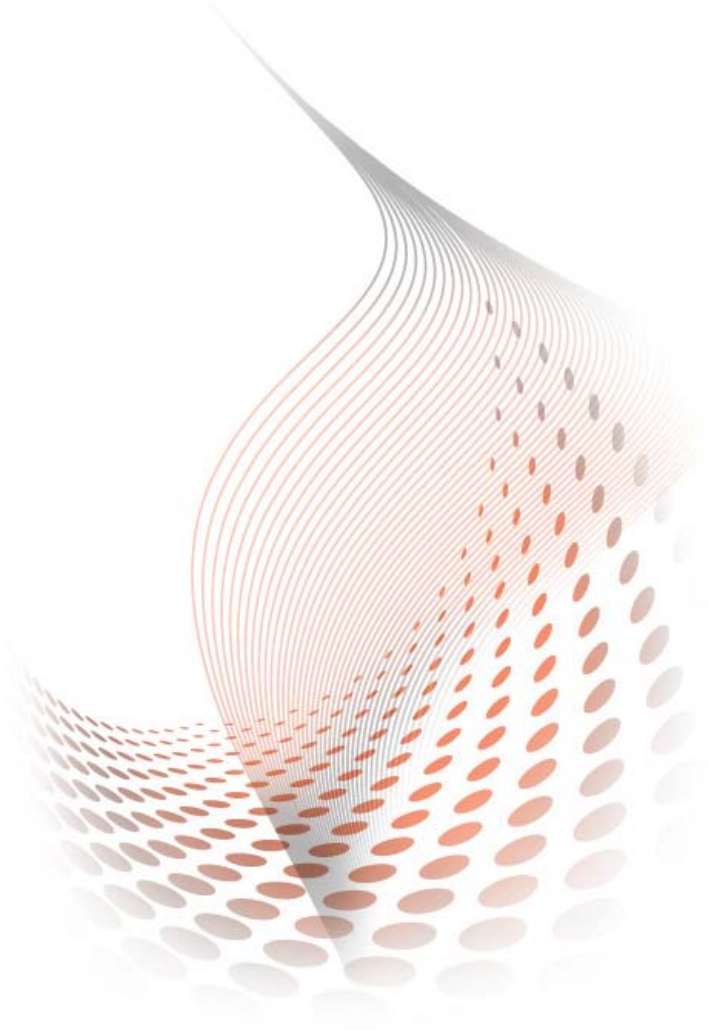
Introduction to Game Consoles Hardware and Computer Graphics

CPU and GPU architectures and more



Game Dev School

Agenda

- Introduction
- Computer Hardware Prehistory
- Hardware Basics
- Programming Languages
- Scripting Languages – LUA
- Game Consoles and Hardware
- Screen Space
  - Aspect Ratios and resolutions
  - Coordinate systems
  - Preserving FOV
- Devices and Head Mounted Displays
- Human Eye
- Color quantization
- Color representation models
  - Additive models – RGB, sRGB
  - Subtractive models – CMYK
- Raster graphics
- Image file formats
  - Lossless Compressed - TGA, PNG
  - Lossy Compressed – JPEG
- Asset Pipeline
- GPU Architecture – Rendering Pipeline

Introduction

Karol Gasiński

- http://www.linkedin.com/
- kuktus@gmail.com

GPU SW Architect @ Apple

(previously ~7 years @ Intel)

KHRONOS Group member

Contributes to OpenGL Specification

- http://www.khronos.org/

Introduction

Projects:

- True Color graphics library for DOS
- Multitasking Operating System Microkernel in ASM
- Multiplatform game engine (Android/BlackBerry/iOS/Windows)
- WolfensteinVR

Worked on mobile versions of:

- Medieval Total War
- Pro Evolution Soccer
- Silent Hill
- Invaders Must Die!

Currently developing:

- Sci-Fi Title TBA

Introduction

Founder:

- WGK - Polish Conference on Computer Games Development
- http://wgk.gd
- 450+ international participants
- ~40 lectures, panels, workshops
- 4 editions away
- 3 full days

Speaker at:

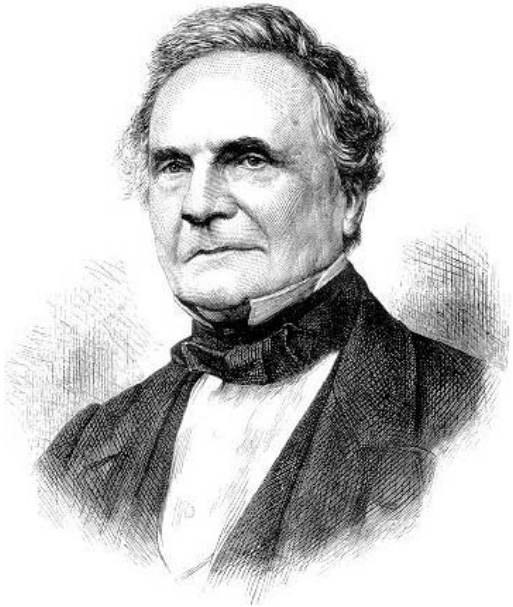
- Digital Frontier, GameDay, WGK, IGK, Gdańsk University of Technology…

You can also meet me at:

- GDC, GDC Europe, SIGGRAPH, GamesCom…


SCORE 0000000 FOOD
LIVES 03
TIME 97
ENERGY:
Prehistory

Computer Hardware Prehistory



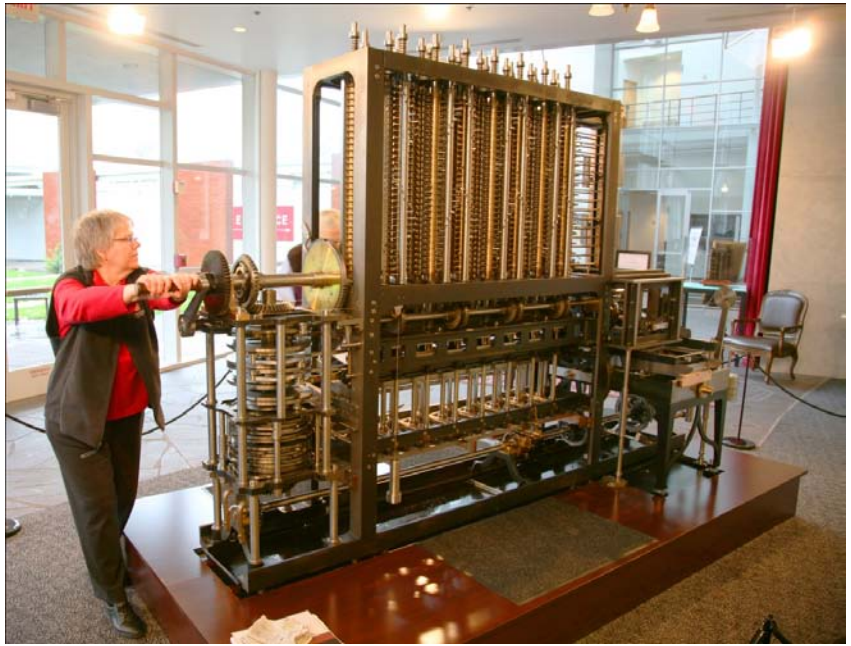
Charles Babbage, 1791-1871

Computer Hardware Prehistory

Computer – człowiek wykonujący obliczenia

1822 – *Maszyna różnicowa (projekt),* miała wykonywać obliczenia do tablic matematycznych dla funkcji wielomianowych.

1837 – *Maszyna analityczna (projekt),* pierwszy programowalny komputer ogólnego przeznaczenia. Miała być mechaniczną konstrukcją napędzaną przez silnik parowy.

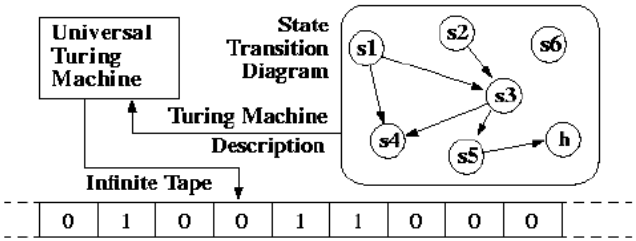


Maszyna różnicowa Babbage'a zbudowana przez zespół z Londyńskiego Muzeum Nauki

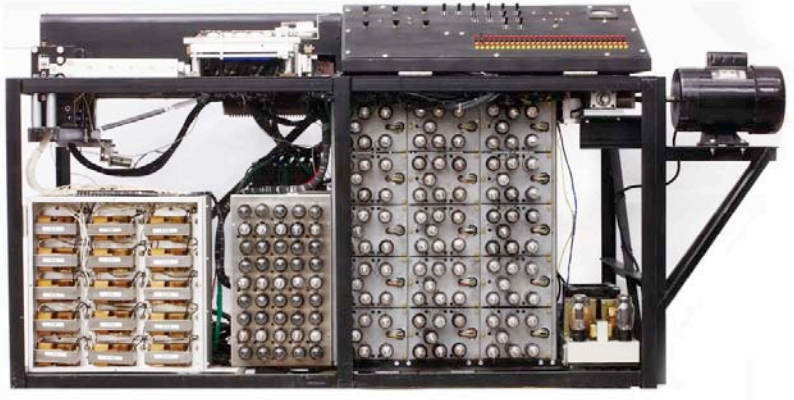
Computer Hardware Prehistory

1930 – Vannevar Bush buduje *Analizator różniczkowy*, najbardziej skomplikowaną mechaniczną maszynę liczącą.

1936 – Alan Turing przedstawia opis *Maszyny Turinga*, pozwalającej sformalizować idee algorytmu i programu

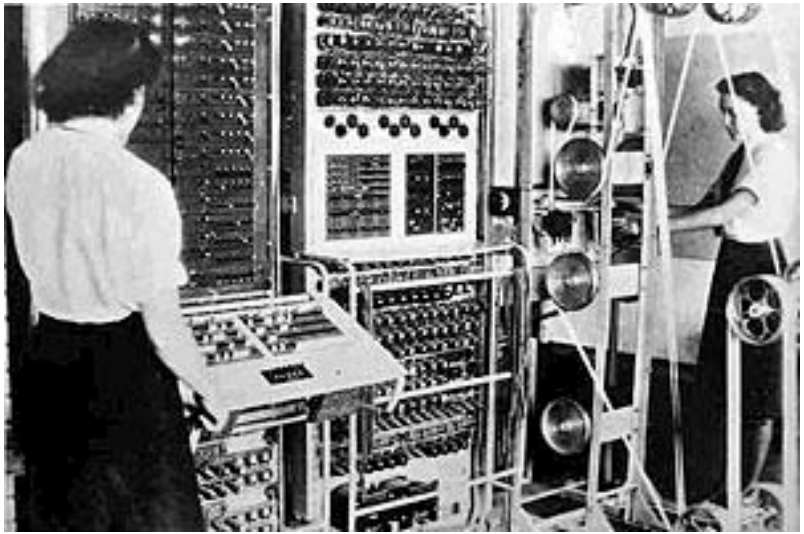


1937 – *Atanasoff–Berry Computer (ABC)* - pierwszy cyfrowy komputer (ale nie programowalny)



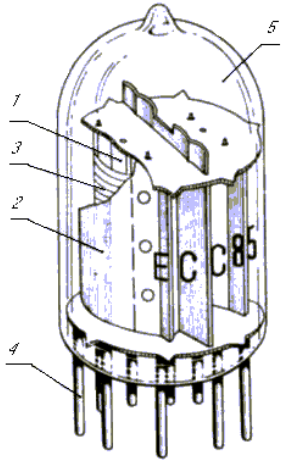
Computer Hardware Prehistory

1943 – *Colossus* - pierwszy cyfrowy w pełni programowalny komputer. Był używany do złamania szyfru Lorenza podczas WWII.



Z początku do budowy komputerów używano mechanicznych przełączników elektromagnetycznych i lamp elektronowych.

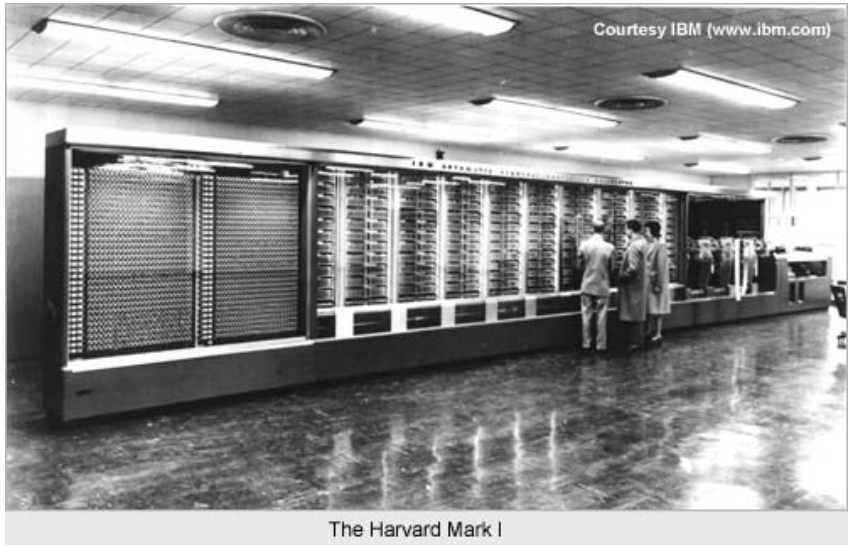
Lampy grzały się jednak i często przepalały, co sprawiało że były dość zawodnym rozwiązaniem.



1) Katoda
2) Anoda
3) Siatka
4) Nóżki
5) Bańka

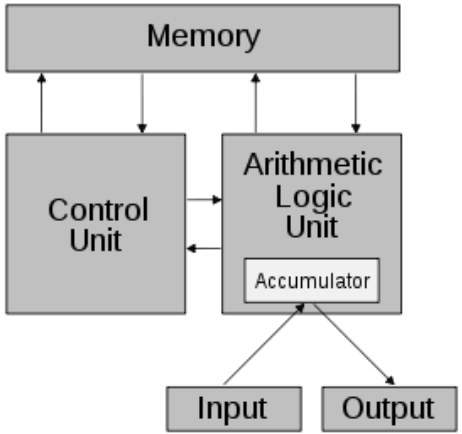
Computer Hardware Prehistory

1944 –*Mark 1* zbudowany we współpracy z IBM



The Harvard Mark I

1945 – *John Von Neumann* - proponuje nową architekturę komputera. *Architektura Von Neumanna* staje się podwaliną współczesnych komputerów.



Memory – przechowuje kod programu i jego dane. Każda komórka pamięci ma identyfikator zwany jej adresem.

Control Unit – Odpowiedzialna za pobieranie instrukcji z pamięci i ich sekwencyjne przetwarzanie.

ALU – Wykonuje operacje arytmetyczne.

I/O – Urządzenia wejścia/wyjścia służą do interakcji z operatorem.

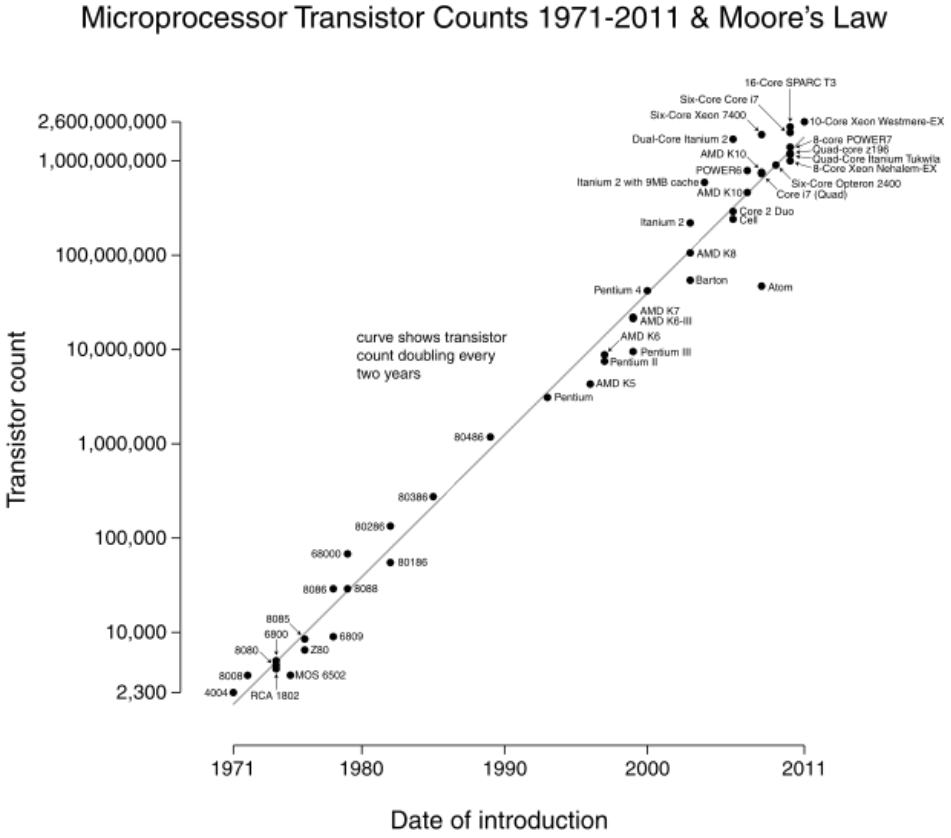
Computer Hardware Prehistory

1947 - John Bardeen, William Shockley i Walter Brattain przeprowadzają pierwsze eksperymenty nad *tranzystorem*.

1954 – Pierwszy krzemowy tranzystor wyprodukowany przez Texas Instruments.

1968 – Gordon Moore i Robert Noyce zakładają firmę Int$_e$l (od Integrated Electronics).
1970 – powstaje prawo Moore’a - co dwa lata ilość tranzystorów ma wzdrastać dwukrotnie.

Computer Hardware Prehistory



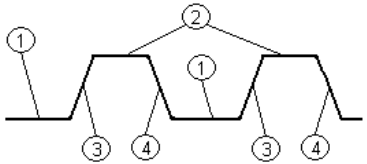
Hardware

Hardware Basics

Podstawą działania komputerów jest zmiana napięcia sygnału. Napięcie niskie (np. 0.8V) odpowiada stanowi spoczynku, czyli „0”.

Napięcie wysokie (np. 1.2V) odpowiada stanowi wysokiemu, czyli „1” (napięcie potrzebne do pobudzenia innych układów)

1) „0” – 0.8V brak pobudzenia układu
2) „1” – 1.2V pobudzenie układu
3) Zbocze narastające
4) Zbocze opadające

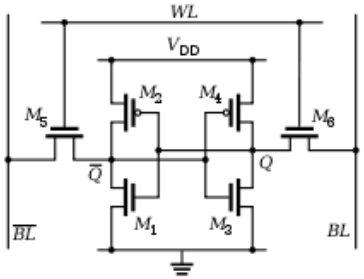


Patrz przedmiot „Układy Cyfrowe”

Ponieważ istnieją tylko 2 stany – „0” i „1”, komputer jest maszyną binarną (dwójkową).

Pojedynczy stan nazywany jest *bitem*.
4 bity tworzą *nibble*.

Za podstawową jednostkę pamięci przyjęło się jednak *bajt*, czyli zbiór 8 bitów.



Pojedyncza komórka pamięci SRAM składająca się z 6 tranzystorów (przechowuje 1 bit).

Patrz przedmiot „Podstawy Elektroniki”

Hardware Basics

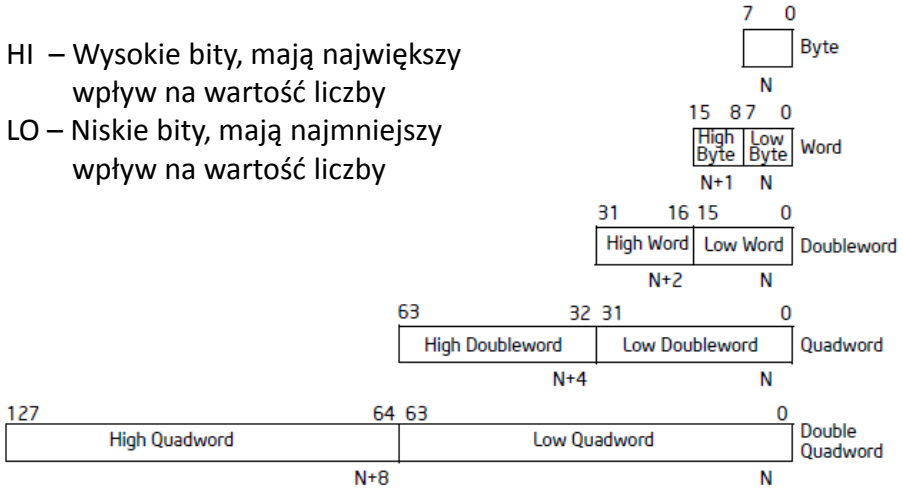
Komputery których jednostki ALU wykonywały obliczenia na bajtach, nazywano 8 bitowymi.
Przykładami takich komputerów są np.:
Commodore 64, ZX Spectrum, Sinclair…

Montezumas Revenge

Hardware Basics

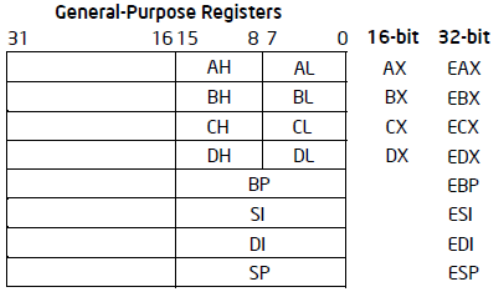
Jednostki opisujące 16 bitowe, 32 bitowe, 64 bitowe i większe obszary danych nazywa się odpowiednio:

HI – Wysokie bity, mają największy wpływ na wartość liczby
LO – Niskie bity, mają najmniejszy wpływ na wartość liczby



Double Quadword jest też zwany OWORD – Octaword

Komputery z rodziny „x86”, są 32 bitowe, czyli posiadają jednostki ALU operujące na 32bitowych rejestrach.

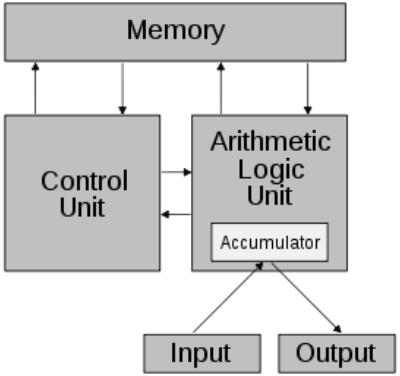


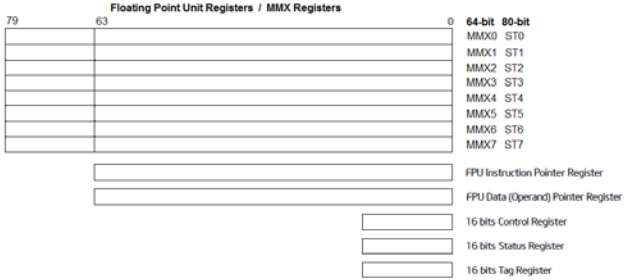
Rejestry przechowują tymczasowe wyniki operacji przed zapisaniem do pamięci.

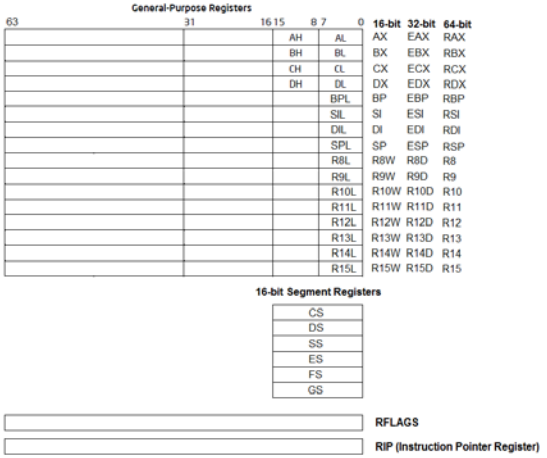
Patrz przedmiot „Architektura Komputerów”

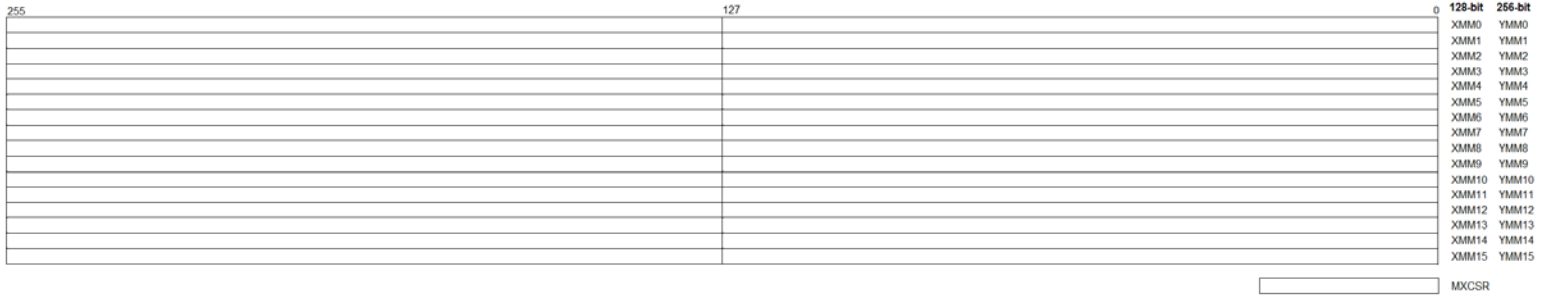
Hardware Basics

Obecnie w sprzedaży są już tylko komputery 64 bitowe. Aktualny zestaw rejestrów poniżej:



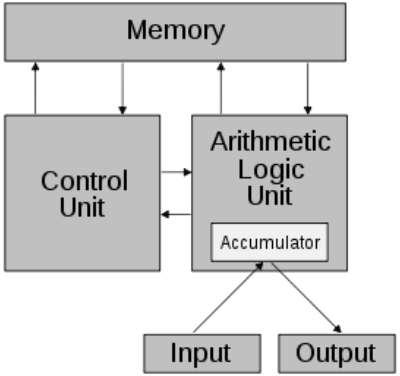


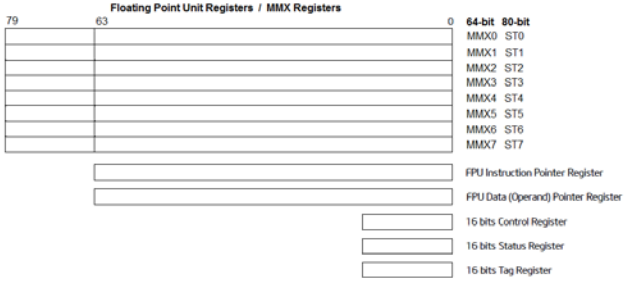


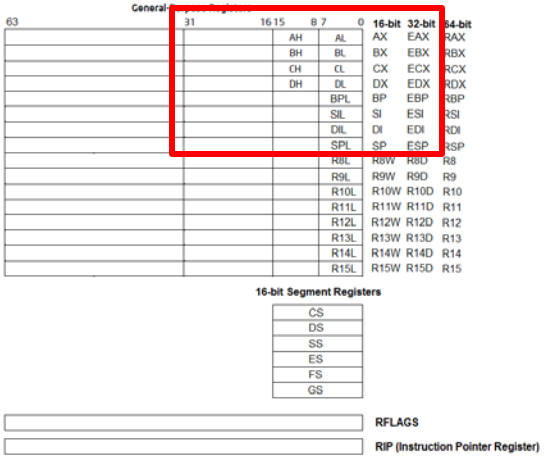


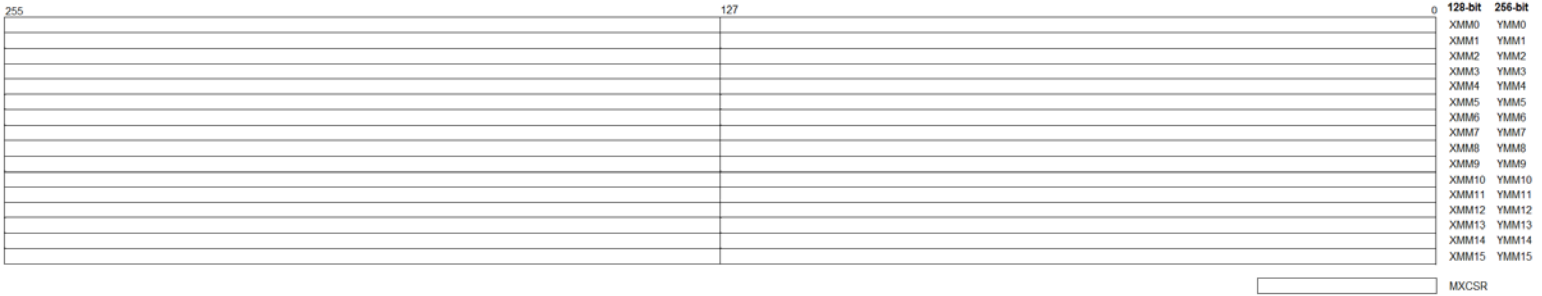
Hardware Basics

Obecnie w sprzedaży są już tylko komputery 64 bitowe. Aktualny zestaw rejestrów poniżej:









Hardware Basics

Ludzie przyjęli system dziesiętny ze względu na ilość palców, np.:

$1984 = (1*10^3) + (9*10^2) + (8 * 10^1) + (4 * 10^0)$

Komputery zmuszone są używać systemu dwójkowego:

$1984 = 1024 + 512 + 256 + 128 + 64 =$

$= (1*2^{10}) + (1*2^9) + (1*2^8) + (1*2^7) + (1*2^6) + (0*2^5) + (0*2^4) + (0*2^3) + (0*2^2) + (0*2^1) + (0*2^0) =$

$= 11111000000b$

11111000000b = 0000 0111 1100 0000 b

Jak widać liczba 1984 będzie przechowana w 16 bitowym rejestrze.

Ponieważ zapis w systemie dwójkowym jest za długi, a zapis w systemie dziesiętnym nieadekwatny, popularny jest również system szesnastkowy:
W systemie szesnastkowym dane opisujemy per nibble:

0000b – 0	0100b – 4	1000b – 8	1100 – C
0001b – 1	0101b – 5	1001b – 9	1101 – D
0010b – 2	0110b – 6	1010b – A	1110 – E
0011b – 3	0111b – 7	1011b – B	1111 – F

Stosujemy prefix „0x” lub postfix „h” :

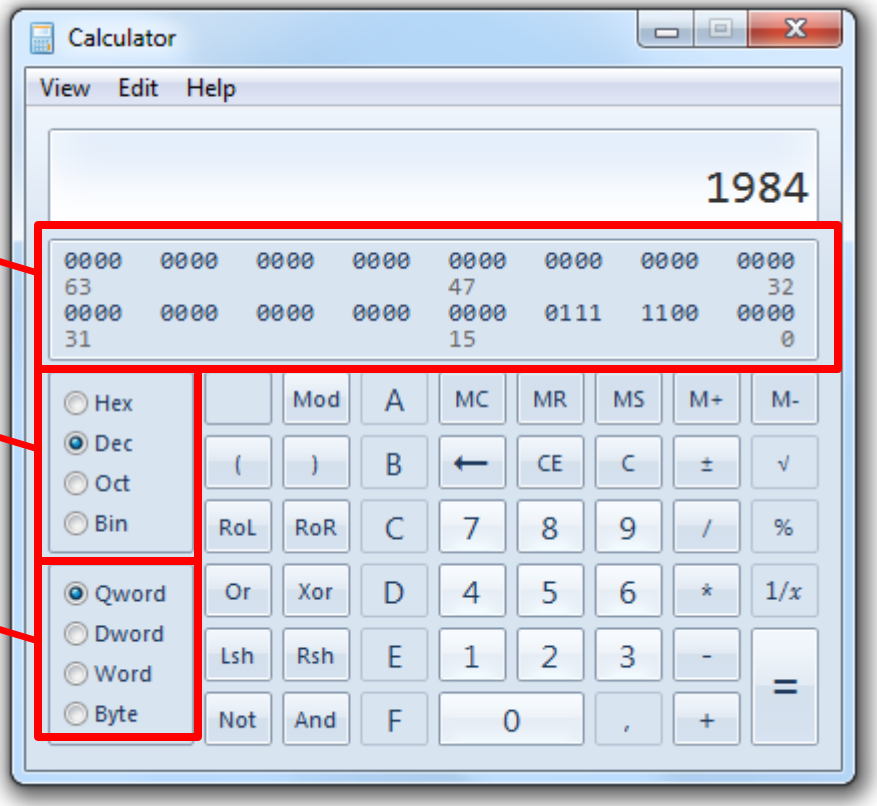
1984 = 11111000000b = 0x7F0 = 7F0h

Hardware Basics

Reprezentacja Binarna w rejestrze

Aktualny system liczbowy

Rozmiar rejestru



Hardware Basics

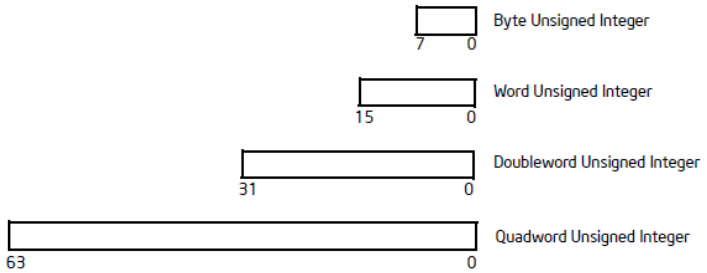
Typy całkowite (*integer*):

[0 … 255]

[0 … 2^{16}-1]

[0 … 2^{32}-1]

[0 … 2^{64}-1]



2^{16}-1 = 65.535

2^{32}-1 = 4.294.967.295

2^{64}-1 = 18.446.744.073.709.551.615

300.000.000.000

(Szacowana ilość wszystkich gwiazd w naszej galaktyce.)

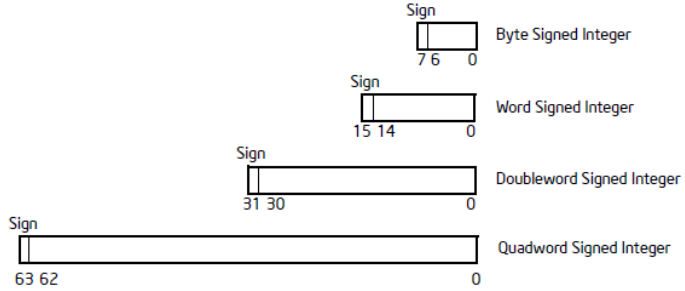
Typy całkowite ze znakiem (*signed integer*):

[-128 … 127]

[-32768 … 32767]

[-2^{31} … 2^{31}-1]

[-2^{63} … 2^{63}-1]



Ostatni, wysoki bit zapalony oznacza liczbę ujemną. W przyjętej arytmetyce brak -0 stąd -128 i „tylko” +127.

11111111b = -1

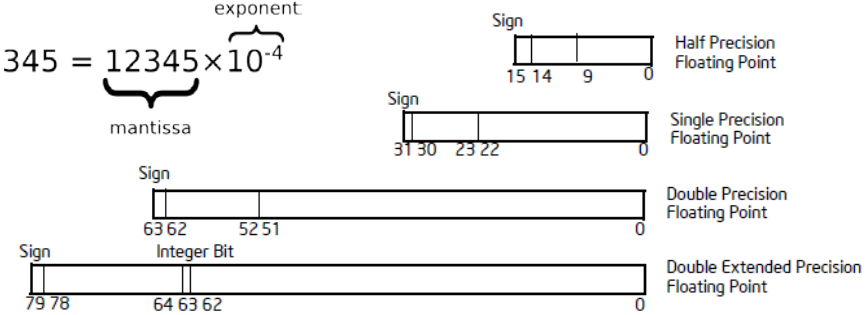
. . .

10000000b = -128

Hardware Basics

Typy zmiennoprzecinkowe (*floating point*):

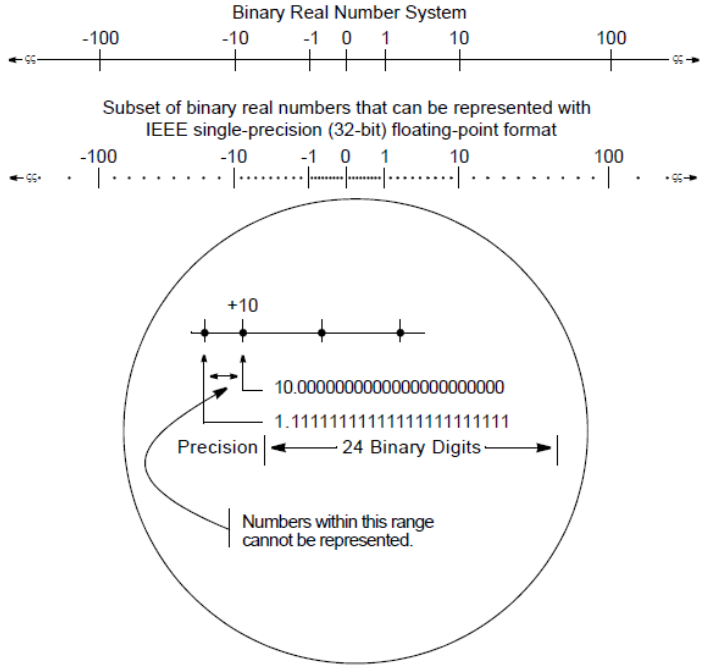
$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \overbrace{10^{-4}}^{\text{exponent}}$$



Bity:	Znak	Exponent	Mantysta	
	1	5	10	Half Precision
	1	8	23	Single Precision
	1	11	52	Double Precision
	1	15	64	Double Extended Precision

3.10×10^{-5} … 6.50×10^{4} Half Precision
1.18×10^{-38} … 3.40×10^{38} Single Precision
2.23×10^{-308} … 1.79×10^{308} Double Precision
3.37×10^{-4932} … 1.18×10^{4932} Double Extended Precision

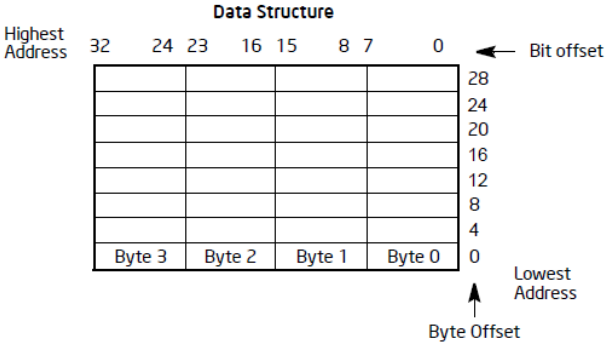
Utrata precyzji:



Patrz przedmiot „Metody Numeryczne”

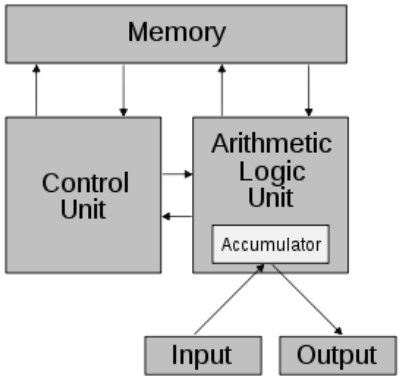
Hardware Basics

RAM (Random Access Memory) - pamięć operacyjna przechowująca programy i dane podczas ich przetwarzania.



Pamięć jest ciągła i odwołujemy się do niej za pomocą liniowych adresów komórek. Możemy ją jednak przedstawiać w dowolny sposób przydatny dla naszych potrzeb.

Jednostką ilości danych jest bajt.



Wielokrotności bajtów					
Przedrostki dziesiętne (SI)			Przedrostki binarne (IEC 60027-2)		
Nazwa	Symbol	Mnożnik	Nazwa	Symbol	Mnożnik
kilobajt	kB	$10^3=1000^1$	kibibajt	KiB	$2^{10}=1024^1$
megabajt	MB	$10^6=1000^2$	mebibajt	MiB	$2^{20}=1024^2$
gigabajt	GB	$10^9=1000^3$	gibibajt	GiB	$2^{30}=1024^3$
terabajt	TB	$10^{12}=1000^4$	tebibajt	TiB	$2^{40}=1024^4$
petabajt	PB	$10^{15}=1000^5$	pebibajt	PiB	$2^{50}=1024^5$
eksabajt	EB	$10^{18}=1000^6$	eksbibajt	EiB	$2^{60}=1024^6$
zettabajt	ZB	$10^{21}=1000^7$	zebibajt	ZiB	$2^{70}=1024^7$
jottabajt	YB	$10^{24}=1000^8$	jobibajt	YiB	$2^{80}=1024^8$

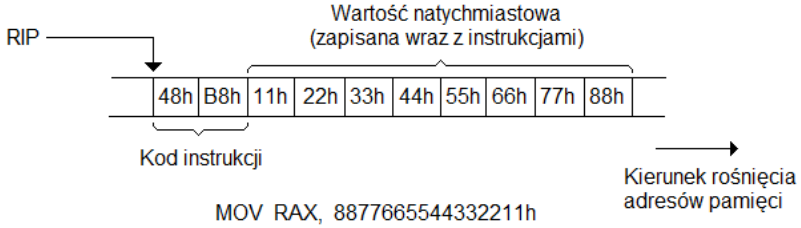
Przy czym w powszechnym użyciu stosuje się przedrostki dziesiętne mając na myśli wartości binarne, czyli:
1KB = 1024B etc.

Programming

Ken Thompson i Dennis Ritchie,
twórcy języka programowania C.

Programming Languages

Rejestr RIP wskazuje adres, w pamięci przechowującej program, spod którego procesor ma odczytać kolejną instrukcję do wykonania np.:



Instrukcje mogą być różnej długości, a ich ciąg tworzy polecenia z których składa się program.

Złożenie (*to assemble*) ciągu instrukcji zrozumiałych dla człowieka w ciąg bajtów tworzących program nazywamy kompilacją.

Program dokonujący tego zadania z najprostszego opisu kodu nazywa się *assemblerem* . Z tego też powodu kod maszynowy tworzący ciąg takich instrukcji jest potocznie również nazywany *assemblerem*.

Zamiana kodu bardziej skomplikowanych języków programowania nazywana jest *kompilacją* .

Programming Languages

Języki kompilowalne to takie których kod jest przetwarzany do kodu maszynowego i może być wykonany bezpośrednio przez procesor.

Zalety:

- Kod wykonywany bezpośrednio na procesorze, bardzo szybki

Wady:

- Każda zmiana w programie, wymaga jego rekompilacji przed ponownym uruchomieniem

Przykłady:

- Assembler, C, C++

Przykładowy kod menedżera pamięci systemu operacyjnego:

```
; Utworzenie PT's i wpisow w nich

   mov edi, [AdrPD]      ; EDI - Wskaz w PD
   xor ecx, ecx          ; Globalny licznik ramek
   mmip_p:
       mov eax, ecx      ;\ Jezeli numer aktualnej ramki
       and eax, 0x3FF    ; |jest ALIGN 1024 to trzeba
       jnz mmip_d3       ;/ zaalokowac nowa PT
           push edi              ;\Odklada wskaz w PD i
           push ecx              ;/licznik ramek
           call MM_AllocFrame    ;\Alokuje nowa PT i uaktualnia
           mov esi, eax          ;/wskaz w PT
           mov edi, eax          ;\ Czysci zaalokowana ramke
           mov ecx, 1            ; |dla pelnego bezpieczenstwa.
           call MM_ClearFrames   ;/
           pop ecx               ;\Przywraca wskaz w PD
           pop edi               ;/i licznik ramek
           mov eax, esi          ;\Pobiera adres nowej PT
           add eax, 3            ;/i tworzy z niego PDE
           mov [edi], eax        ;\Montuje PT w PD i przesuwa
           add edi, 4            ;/wskaz w PD na kolejne pole
       mmip_d3:
       mov eax, ecx      ; Pobiera numer aktualnej ramki
       shl eax, 12       ;\Na podstawie numeru ramki
       add eax, 3        ;/tworzy jej PTE
       mov [esi], eax    ; Zapisuje PTE ramki
       add esi, 4        ; Przesuwa wskaz w PT
       inc ecx           ; Przesun numer na kolejna ramke
   cmp ecx, [Frames]     ;\Jezeli nie zainicjowal wszystkich
   jb  mmip_p            ;/ramek to zapetla
```

Historia języków programowania:
http://www.levenez.com/lang/lang_a4.pdf

Programming Languages

Programowanie w assemblerze jest żmudną pracą a kod jest wrażliwy na błędy. Dlatego w 1954 powstał *Fortran*.

Język C został stworzony z myślą o pisaniu systemów operacyjnych na początku lat 70. Umożliwia on szczegółowe zarządzanie zasobami komputera dzięki czemu programy w nim napisane są równie szybkie co te napisane w assemblerze.

Jest to jeden z głównych powodów dla których jest on wybierany przez twórców gier komputerowych – pełna kontrola nad sprzętem.

C wymagają określenia typów dla zmiennych których chcemy używać:

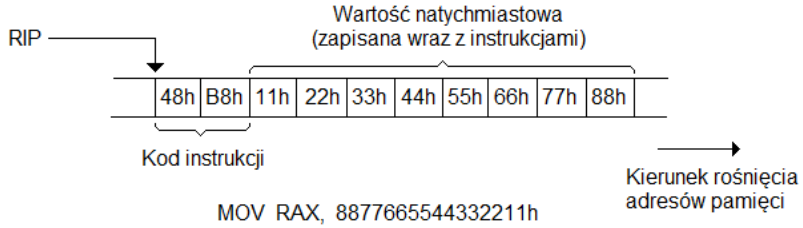
char	– 8 bit Byte Signed Integer
short	– 16 bit Word Signed Integer
int	– 32 bit Doubleword Signed Integer
long long int	– 64 bit Quadword Signed Integer
float	– 32 bit Single Precision Floating Point
double	– 64 bit Double Precision Floating Point
long double	– 80 bit Extended Double Precision Floating Point

Przedrostek „unsigned” oznacza zmienną przechowującą liczby całkowite bez znaku np.:

unsigned int	– 32 bit Doubleword Integer

Programming Languages

Wracając do naszej instrukcji assemblerowej:



Assembler:

MOV RAX, 8877665544332211h

C++:

unsigned long long int zmienna;
zmienna = 0x8877665544332211;

Oznacza to jednak, że język C jest silnie typowany (*strongly typed*) i trzeba zwracać uwagę na wiele detali.

Weak Typed

```
a = 2
b = "2"
concatenate(a, b)      # Returns "22"
add(a, b)              # Returns 4
```

Java Script, Perl, PHP, LUA

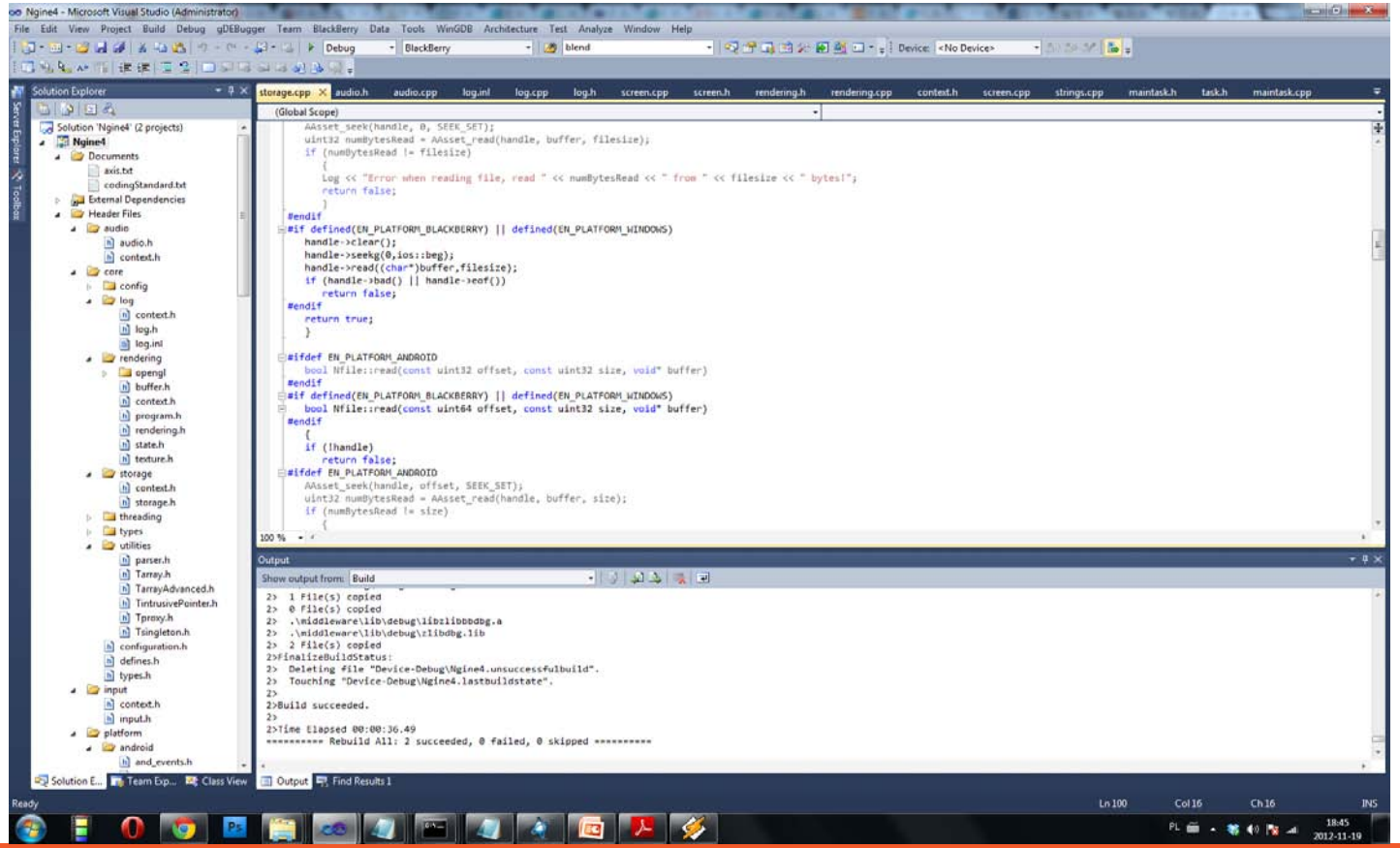
Strongly Typed

```
a = 2
b = "2"
concatenate(a, b)      # Type Error
add(a, b)              # Type Error
concatenate(str(a), b) # Returns "22"
add(a, int(b))         # Returns 4
```

Action Script 3, C, C++, C#, Java

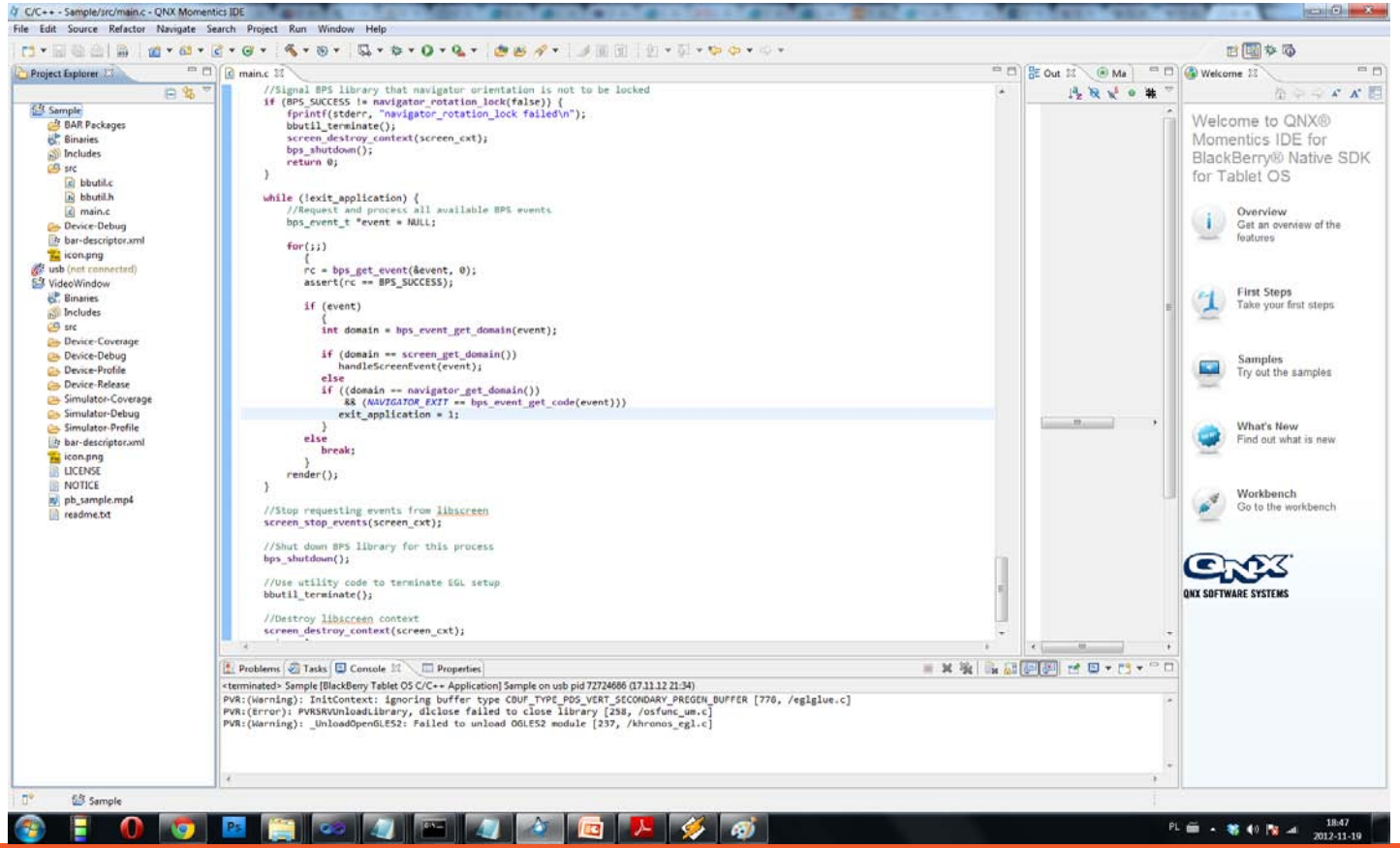
Początkujący programiści wolą języki słabo typowane w których kompilator sam domyśla się typu danych (jeżeli jest to możliwe).

Programming Languages

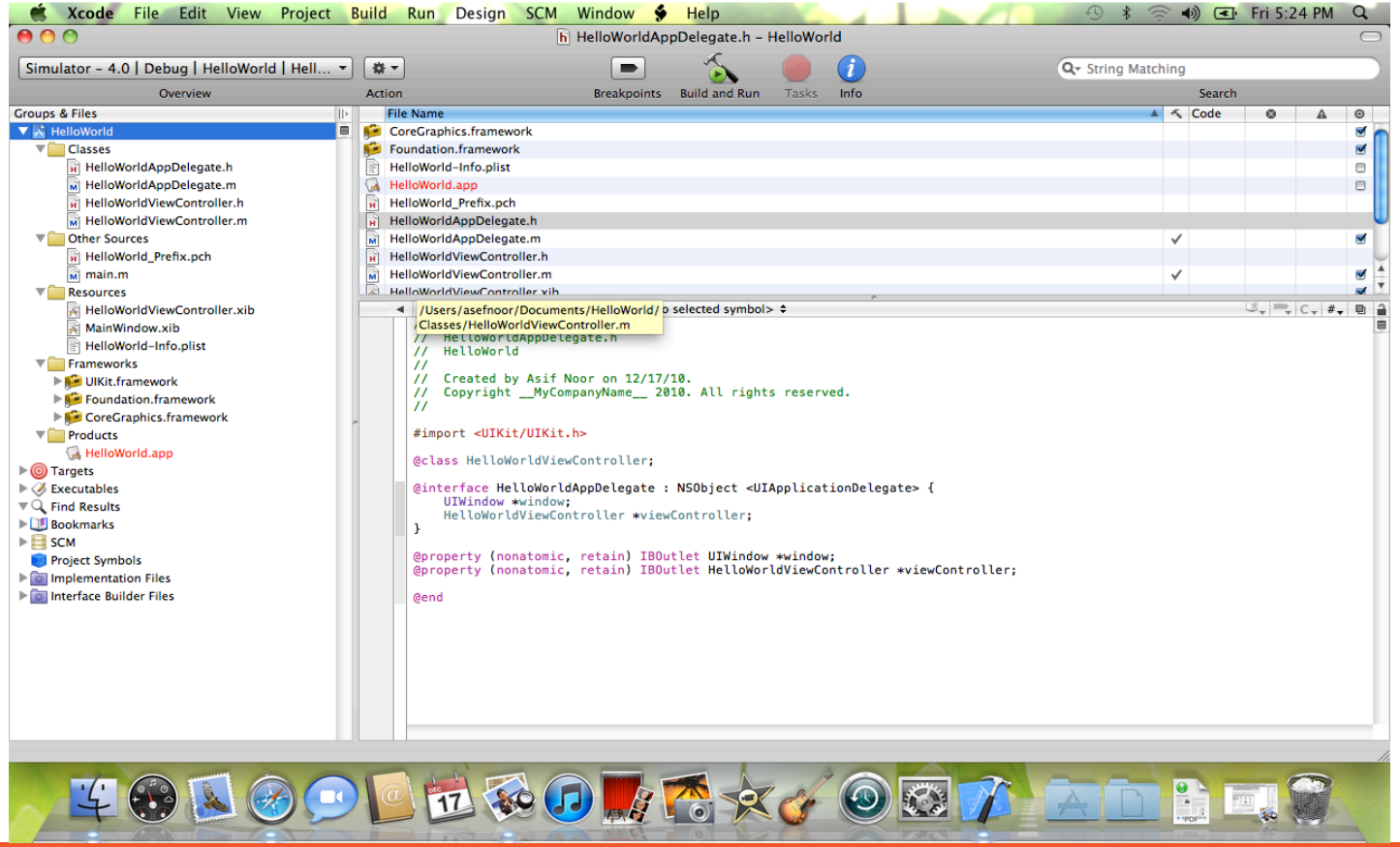


Game Dev School

Programming Languages



Programming Languages



Scripting Languages - LUA

Języki skryptowe to takie, których kod zwany też skryptem, uruchamiany jest na procesorze za pośrednictwem innego programu w *maszynie wirtualnej* .

Zalety:

- Zmiany w skrypcie nie wymagają rekompilacji.
- Możliwe jest edytowanie skryptów „w locie”.

Wady:

- Skrypty są wolniejsze w wykonaniu niż kod natywny z powodu intrpretacji w czasie rzeczywistym.

LUA jest najpopularniejszym językiem skryptowym stosowanym w GameDevie.

Maszyny wirtualne LUA są implementowane w wielu silnikach do tworzenia gier.
Innymi językami skryptowymi są np.:

- Quake C, UnrealScript, GameMonkey, AngelScript, Python...

LUA jest językiem luźno typowanym więc możemy po prostu napisać:

```
zmienna = 0x887766554433221l
```

Scripting Languages - LUA

W LUA istnieje osiem abstrakcyjnych typów danych:

nil - nic, zwracany np. w przypadku błędu, brak wyniku
boolean - zmienna boolowska, *true* lub *false*
number - liczba dowolnego typu
string - ciąg znaków
function - funkcja
userdata - wskaźnik na obszar pamięci (C)
thread - wątek
table - tablica danych

Najprostrzy program w LUA:

```
print(„Hello World”)
```

„Hello World” w kilkudziesięciu językach programowania:
http://en.wikipedia.org/wiki/Hello_world_program_examples

Nie musimy deklarować żadnych nagłówków, ani miejsca rozpoczęcia programu. Skrypt LUA jest interpretowany linijka po linijce i gdy zostanie znalezione polecenie do wykonania, zostanie ono wykonane.

Możemy skomentować fragment kodu:

```
-- To jest komentarz zajmujacy jedna linie

--[[-- To jest dłuższy komentarz który
szczegółowo opisuje zasadę działania
programu i wymaga kilku linii --]]--
```

Komentarze są pomijane podczas interpretacji skryptu.

Scripting Languages - LUA

Programy składają się w większości z różnych warunków które wykonują się lub też nie w zależności od wyników poprzednich operacji.
Najprostrza instrukcja warunkowa to *if* :

```
a = 5
b = 3
if a ~= b then
        print(„A jest rozne od B”)
end
```

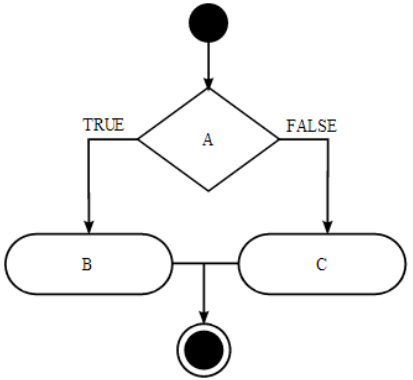
Operatory porównania:

==	równość	<=	mniejszy lub równy
~=	nierówność	>=	większy lub równy
not	negacja	<	mniejszy
		>	większy

Instrukcja warunkowa *if –else* :

```
a = 5
b = 3
if a ~= b then
    print(„A jest rozne od B”)
else
    print(„A jest rowne B”)
end
```

```
if A then
  B
else
  C
end
```

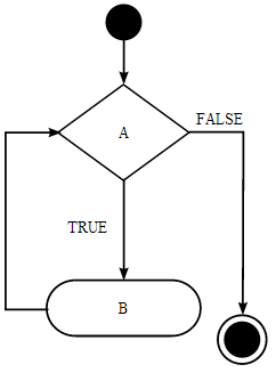


Scripting Languages - LUA

Instrukcja warunkowa *while –do* :

```
a = 0
while a < 2 do
    print(„hello!”)
    a = a + 1
end
```

```
while A do
  B
end
```

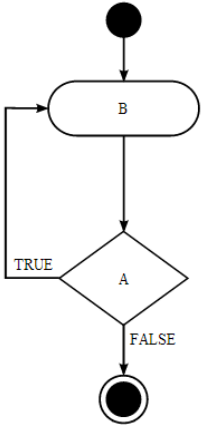


Instrukcja warunkowa *repeat –until* :

```
a = 0
repeat
    print(„hello!”)
    a = a + 1
    if ( a == 4 )
        break
until a < 5
```

```
repeat
B
until A
```

Chcemy wykonać ciało pętli minimum raz.



Operatory porównania:

==	równość	<=	mniejszy lub równy
~=	nierówność	>=	większy lub równy
not	negacja	<	mniejszy
		>	większy

Wykonanie aktualnej pętli można przerwać z dowolnego miejsca w jej wnętrzu za pomocą polecenia *break* .

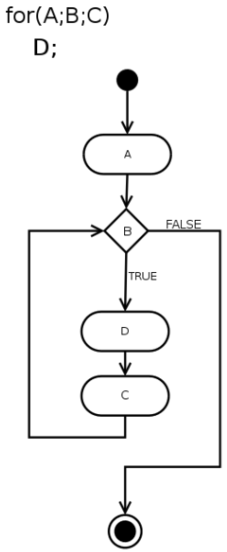
Scripting Languages - LUA

Instrukcja pętli *for* :

```
for i=1, 10, 2 do
    print(i)
end
```

Wynik:

1
2
5
7
9



Operatory matematyczne:

= przypisanie
+ suma
- różnica
***** mnożenie
/ dzielenie
^ potęgowanie (Shift+6)
% modulo, np:
print(14 % 5) -- Zwraca 4
.. Konkatenacja „I”..”am”..”Legend” da w rezultacie „IamLegend”
Długość, np:
tablica = {"do", "re", "me"}
print("Mam "..#tablica.." rzeczy") -- Mam 3 rzeczy

Instrukcja for ma składnie:

for number, maximum (number < maximum), step

Scripting Languages - LUA

Możemy też deklarować tablice danych:

```
tablica = {}
table.insert( tablica, 7 )
table.insert( tablica, 5 )
table.sort( tablica )
table.foreach( tablica, print )
```

Tablice są indeksowane od 1 .. N

```
print( tablica[0] ) -- Rezultat: nil
```

Do operacji na tablicach potrzebujemy odpowiednich poleceń.

Bardziej złożone programy składają się z funkcji, które grupują kod w logiczne bloki.

Funkcje mogą:

- przyjmować parametry
- zwracać wyniki
- wywoływać siebie oraz inne funkcje nawzajem

Przykład funkcji:

```
function nazwa_funkcji( nazwa_parametru )
    ...
    return zwracany_wynik
end

wynik = nazwa_funkcji(5)
```

Scripting Languages - LUA

Zmienne lokalne istnieją tylko w ramach aktualnego bloku instrukcji, np. w funkcji:

```
function hello( imie )
    local nazwisko = ” Lock”
    print( ”Hello ”..imie..nazwisko )
end

hello( „John” )
```

Rezultat:

```
Hello John Lock
```

Zbiór funkcji nazywamy biblioteką. Biblioteki w LUA można tworzyć zapisując zbiór funkcji w pliku z rozszerzeniem „lua”.

Bibliotekę taką można załadować w następujący sposób:

```
loaded_chunk = assert(loadfile(„biblioteka.lua"))
loaded_chunk()   -- Inicjacja funkcji z biblioteki
foo(2, 3)        -- Wywolanie funkcji z biblioteki
```

Poza funkcjami zadeklarowanymi przez programistę, istnieją gotowe biblioteki dostarczane w ramach maszyny wirtualnej. Zestawy te można podzielić na dwie kategorie:

- Standardowe funkcje maszyny wirtualnej:
  - print, math, loadstring, string, table ...
- Funkcje interfejsowe silnika udostępniające jego wewnętrzne mechanizmy.

Game Consoles

Aspect Ratio

- Display physical width to height proportion. For consistency we always specify it in landscape view (when dealing with smartphones).
- In good old days all displays had 4:3 aspect ratio.
- With introduction of HDTV and mobile devices full range of new aspect ratios emerged.

Game Consoles and Hardware

Desktop and Laptop Aspect Ratio examples:

- Classic 4:3 resolutions:
  - 640x480,
  - 800x600,
  - 1024x768,
  - 1600x1200, etc.
- HDTV 16:9 resolutions:
  - 1280x720 (720p),
  - 1366x768,
  - 1920x1080 (1080p),
  - 2560x1440
  - 3840x2160 (2160p - 4K Ultra HDTV)
  - 7680x4320 (4320p – 8K Ultra HDTV)
- Extended HDTV 16:10 resolutions :
  - 1920x1200, 2560x1600

And when you think you’re ready for everything:
Cinematic LG 29” 2560x1080 21:9

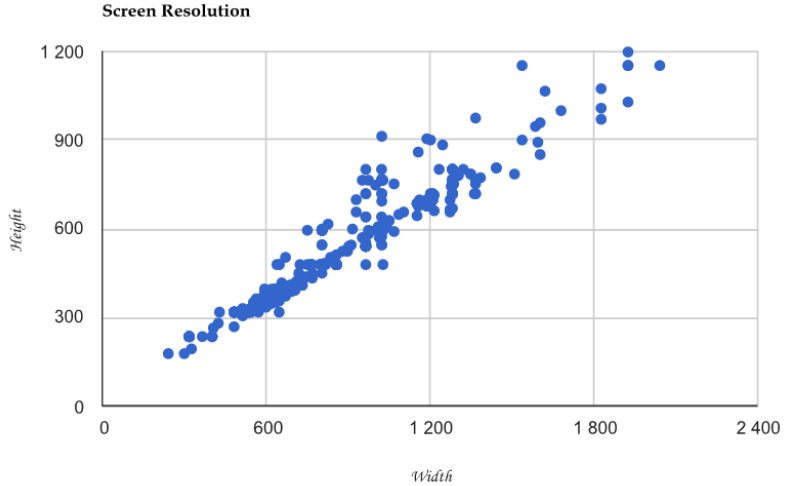
Game Consoles and Hardware

Most popular mobile devices Aspect Ratio and resolution:

Device / OS	Aspect Ratio	Resolution
iPhone+	1.5:1	480x320
iPhone4+	1.5:1	960x640
iPhone5+	almost 16:9	1136x640
iPad1,2,Mini	4:3	1024x768
iPad3+	4:3	2048x1536
BlackBerry 10	16:9 1:1	1280x720 720x720
Nexus 5	16:9	1920x1080
Nexus 10	16:10	2560x1600

Game Consoles and Hardware

Screen resolutions for different mobile devices.
Most common aspect ratio is 5:3.



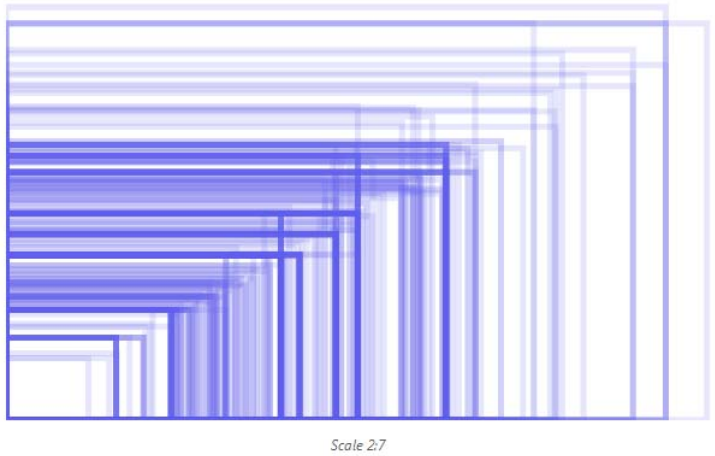
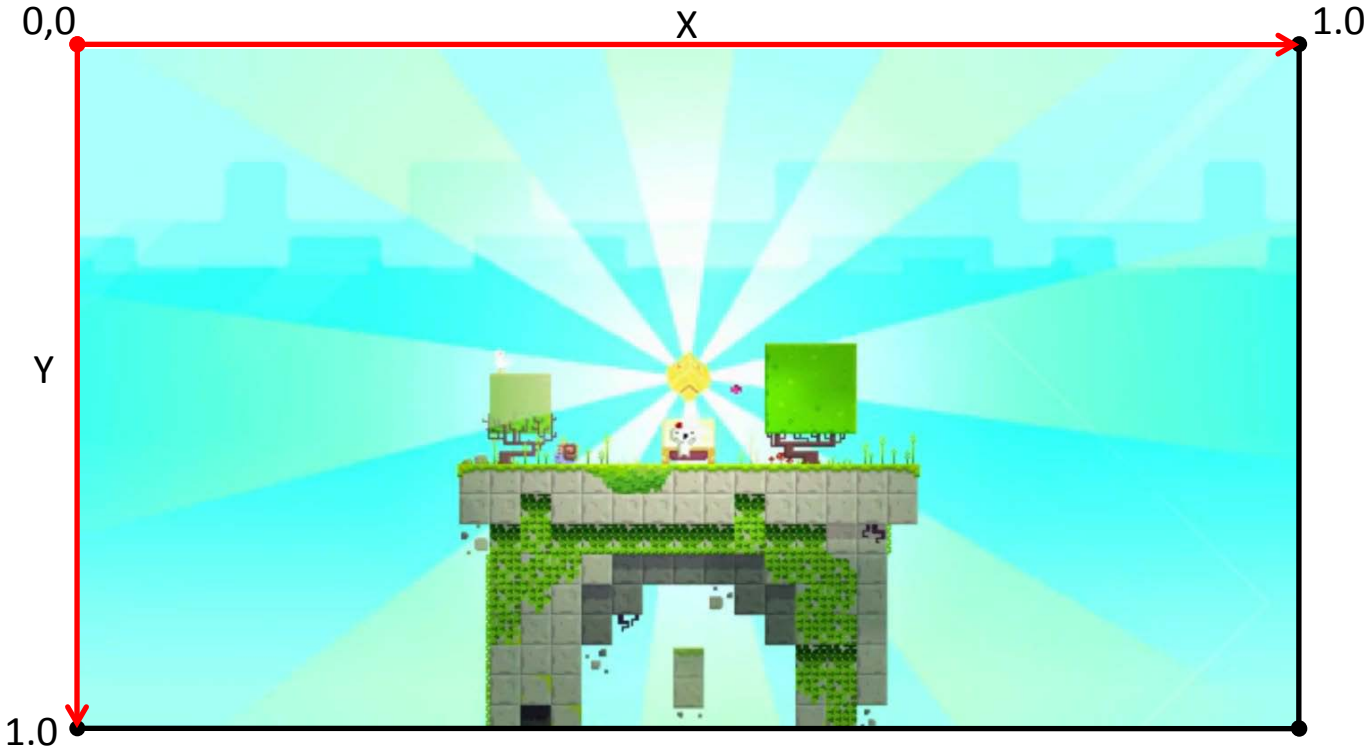


Image is courtesy of OpenSignalMaps

Classic coordinate system

- Good when designing game for one concrete display/console with arbitrary given resolution
- Easy positioning of elements
- Done like in old days
- Per-pixel accuracy
- Cannot be used when designing multiplatform game, PC game or game for range of devices.
- So in fact cannot be used in most cases ☺

Normalized coordinate system



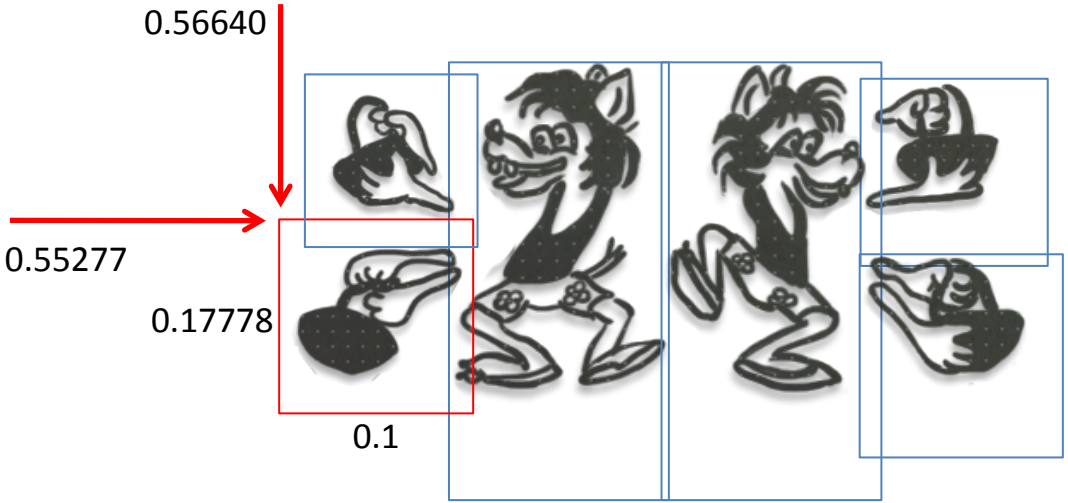
Normalized coordinate system

- Different way of thinking when designing layout
- Screen resolution is not so important
- We position our sprites, user interface elements etc. in normalized range [0.0 … 1.0] in both axes
- Position is global, independent and of any accuracy
- Compatible with future displays and resolutions
- Ideal for multiplatform games
- We still should take into notice aspect ratio but we’re sure that game will be playable without that

Example: Classic USSR handheld games „Nu Pogodi!”

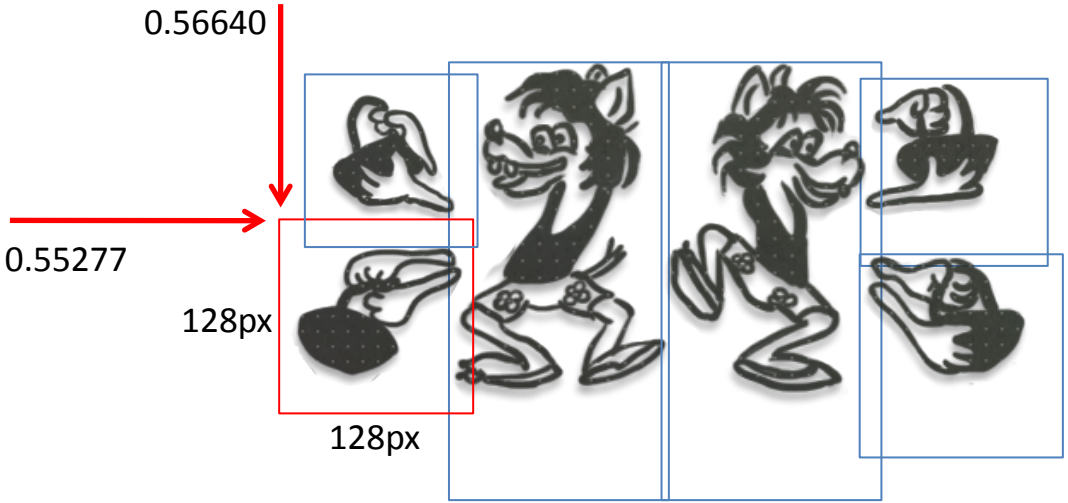
Assets

- Sliced display area into separate transparent parts
- Each part get's it's position, width and height



Assets

- Each asset is stored in PNG with transparency
- Width and height of each asset is power of two



Result

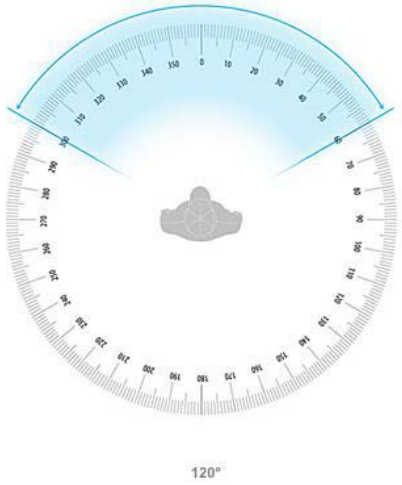

NO LOGO HERE
CATCH'EM EGGZ!
108
PLAY
PAUSE
SOUND
GAMES 4 BLACKBERRY.COM

Game Consoles and Hardware

Result

FOV (*pol. pole widzenia*)

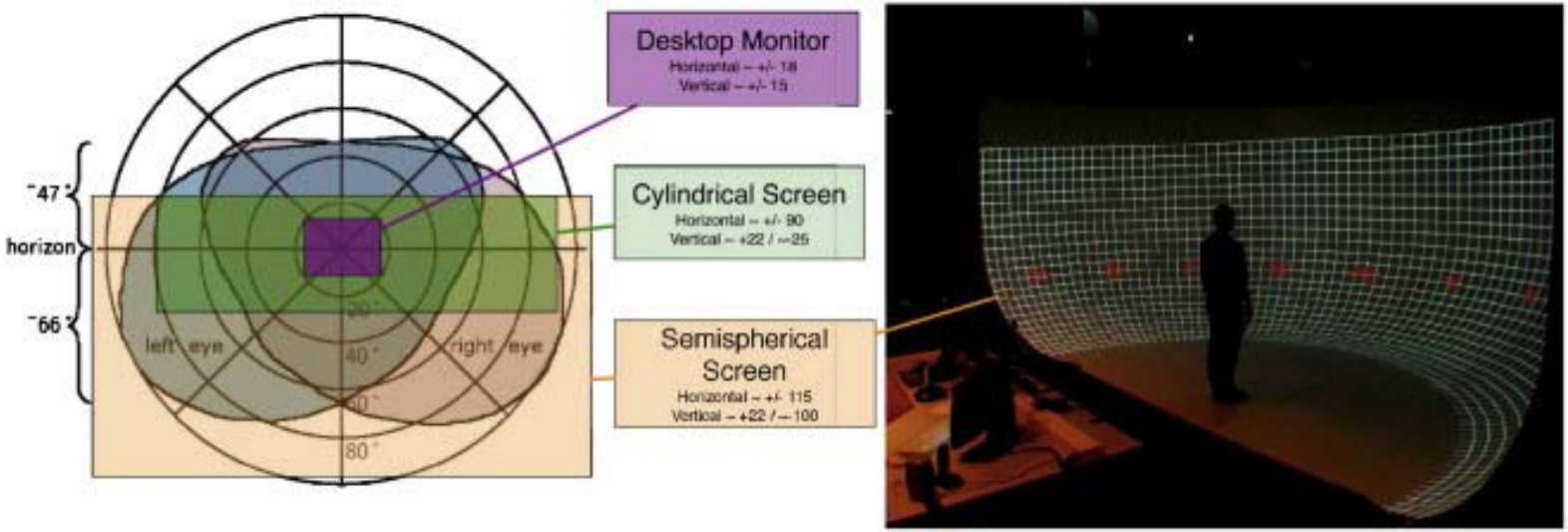
Field of view in both, horizontal and vertical directions (by FOV we mean horizontal FOV)



FOV (horizontal)
~150º Per eye
~120º Stereoscopic, center covered by both eyes
~180º total

Vertical FOV

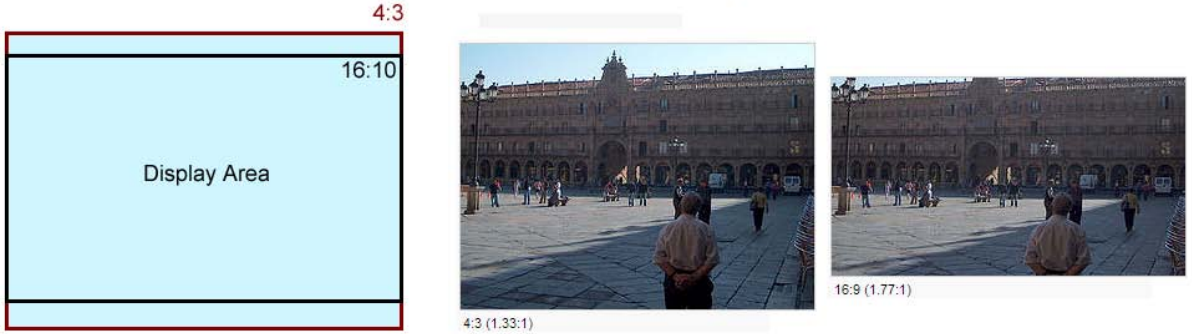
- Nominal line of sigh is 15^{o} below horizon line
- We focus only on area of +/- 15^{o} from that line (from horizon line to -30 degree)
- ~100^{o} total



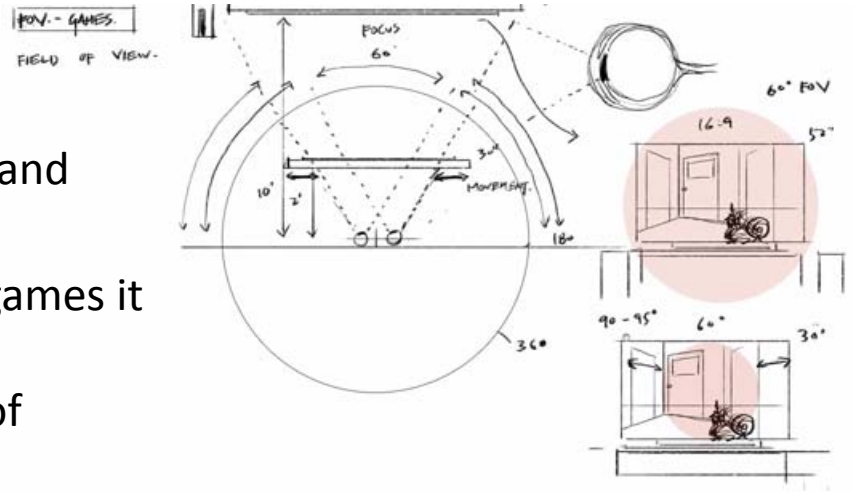
Preserving proper FOV

Correct Relative Areas - Vertical FOV is constant, Horizontal FOV is adjusted dynamically.

Incorrect Relative Areas - Vertical FOV is changing while Horizontal FOV stays thesame.



Adjusting FOV



- FOV depends from display aspect ratio, size and distance from viewer.
- For PC standard is 90º FOV and for console games it is set to 60º. This ensures that in both cases displayed content will cover the same area of Human FOV.

More explanation at:

- http://www.youtube.com/watch?v=blZUao2jTGA

Game Consoles and Hardware

Resolution	Dev-Kit 1	DK2	CV1 (predicted)
Display	1280x800	1920x1080	2560x1440
Per Eye	640x800	960x1080	1280x1440
Field Of View	90	110-130	130
Refresh Rate	60Hz	60Hz, 72Hz, 75Hz	90Hz

Oculus Rift (DK1)

Oculus HD

Oculus Crystal Cove

Oculus DK2

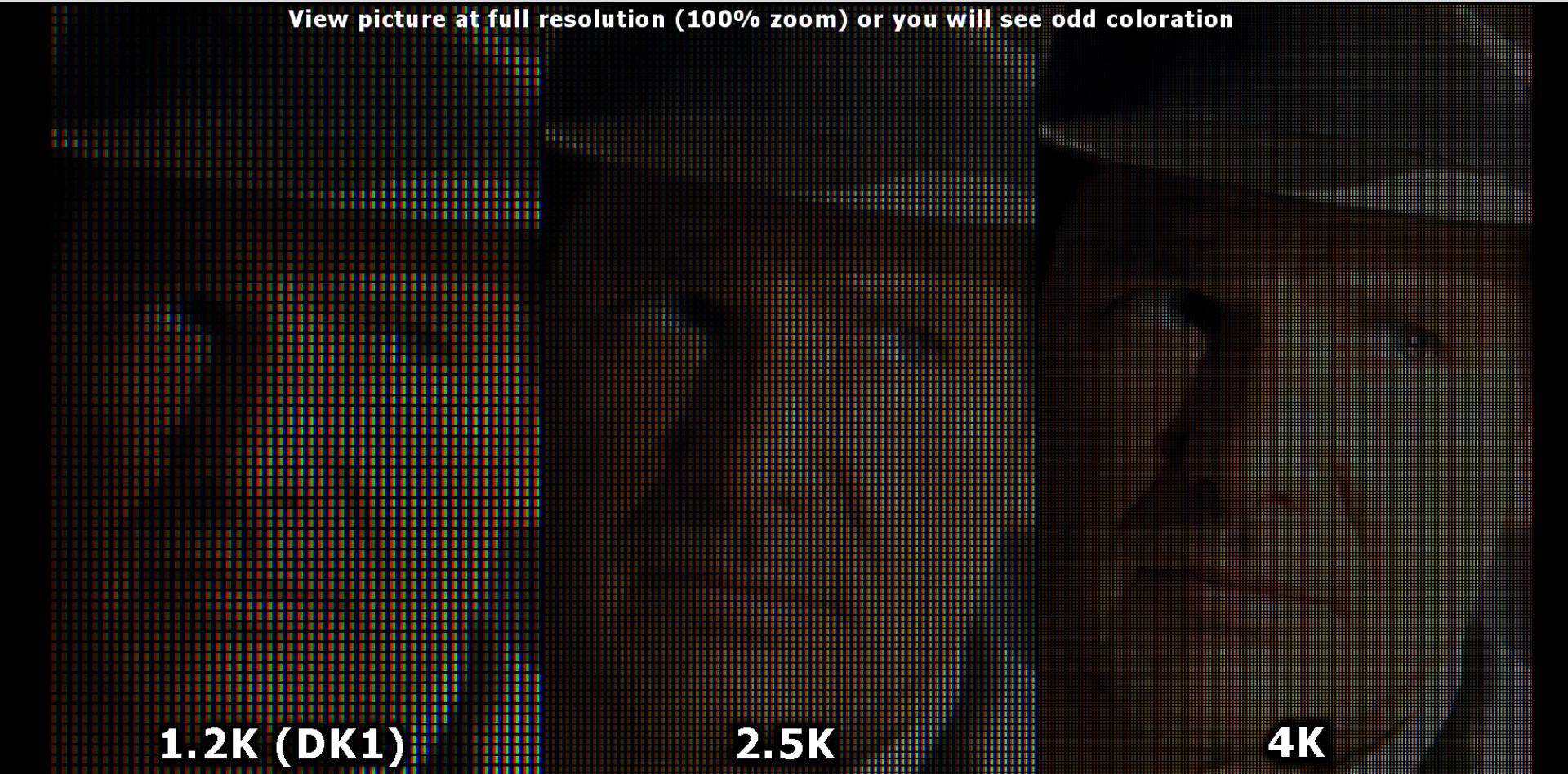
Game Consoles and Hardware

Resolutions comparison:

- Higher density display reduces screen-door effect (effect of visible dark spaces between pixels on display)
- DK2 display greatly increases immersion
- Still required higher PPI display for good quality
- Displays should be longer to cover properly eye field of view.

Resolution	Dev-Kit 1	Dev-Kit 2
Display	1280x800	1920x1080
Per Eye	640x800	960x1080
Visible area	550x550	800x800
Undistorted area	450x450	700x700

Game Consoles and Hardware



Game Consoles and Hardware

– Valve/HTC Vive:

– 1200x1080 resolution per eye (custom/cut FHD displays?), 90Hz
– 2 base laser tracking stations, 2 controllers (15x15ft area)
– Free DevKits only for registered companies.

Game Consoles and Hardware

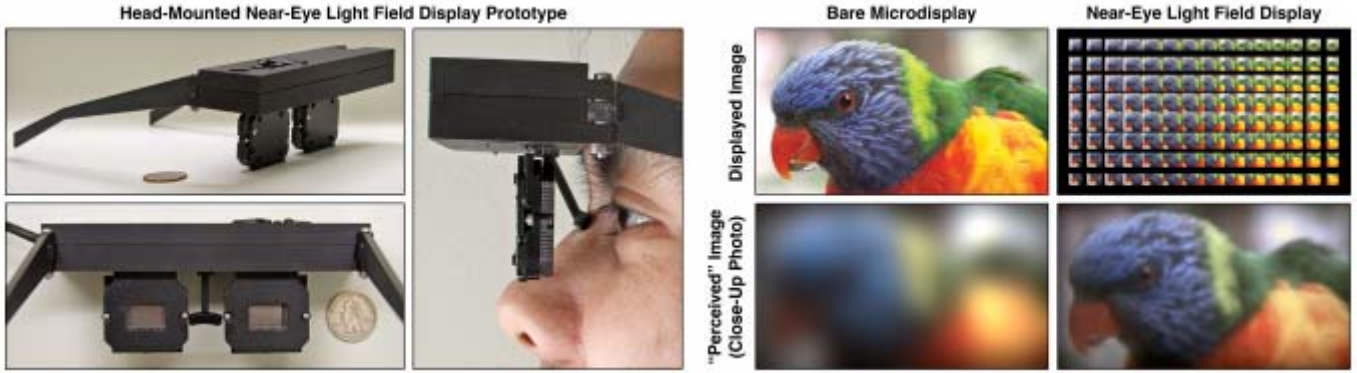
– GearVR Innovator Edition (+ Galaxy Note 4):

– 2560x1440 resolution, only 60Hz
– No positional tracking.

NVidia VR helmet 3D display prototype:

Game Consoles and Hardware

- Software correction for eye vision defect.
- No need for different lenses, one lens set for all observers!
- Very thin displays – 1cm total with lens and case.
- Still low resolution, similar to first Oculus Rift DK1.

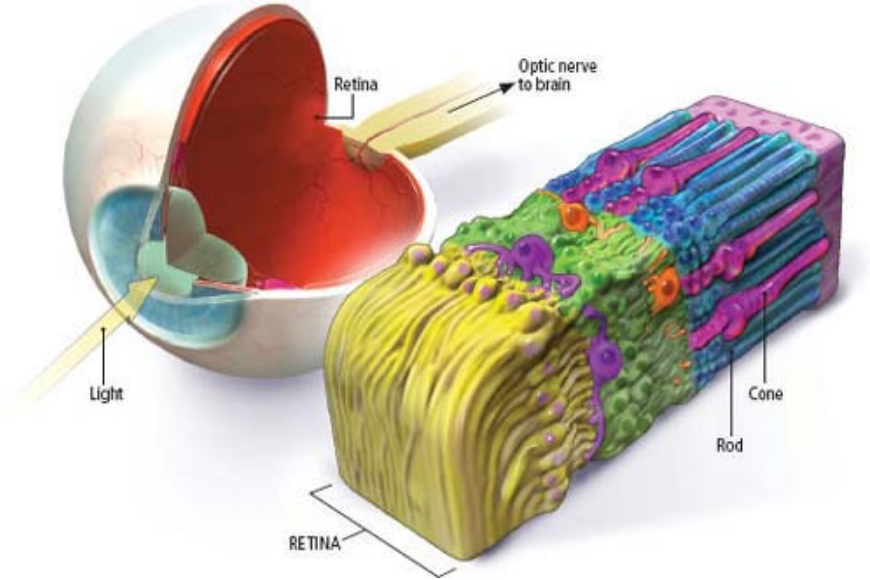


- More details:
  https://research.nvidia.com/publication/near-eye-light-field-displays

Human Eye

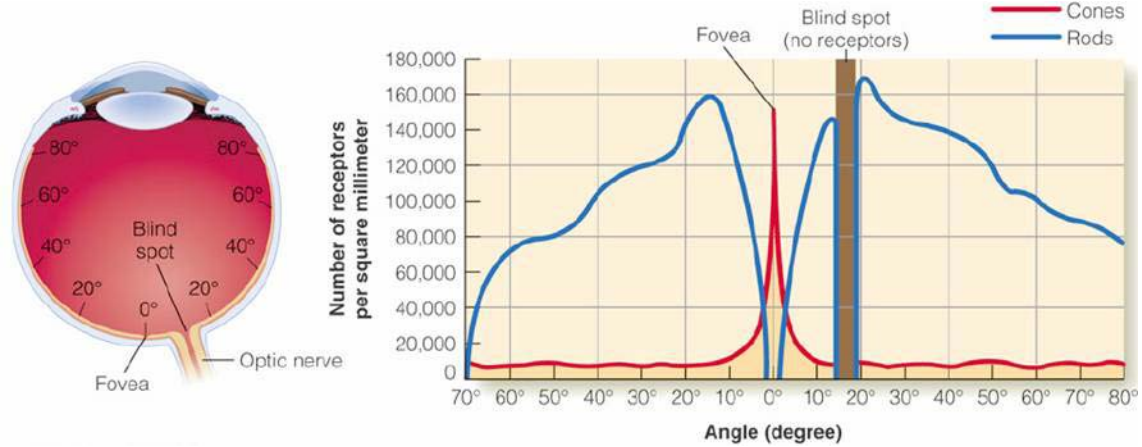
130 mln – Rods (*pol. pręciki*)

7 mln – Cones (*pol. czopki*)



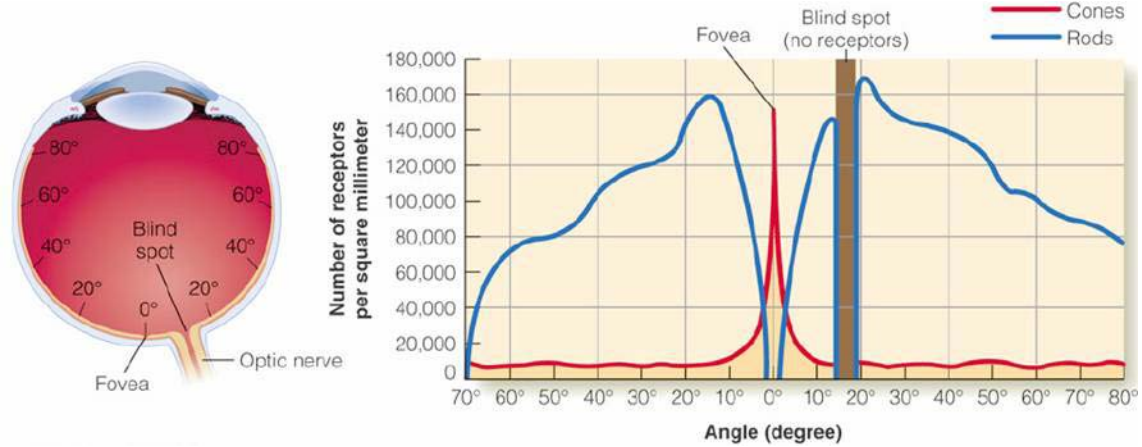
Rods (*pol. pręciki*) are responsible for:

- Recognition of objects outlines and edges
- Sensitive to light intensity
- Orientation in space
- Rim view



Cones (*pol. czopki*) are responsible for:

- High resolution view
- Colors recognition
- Center view



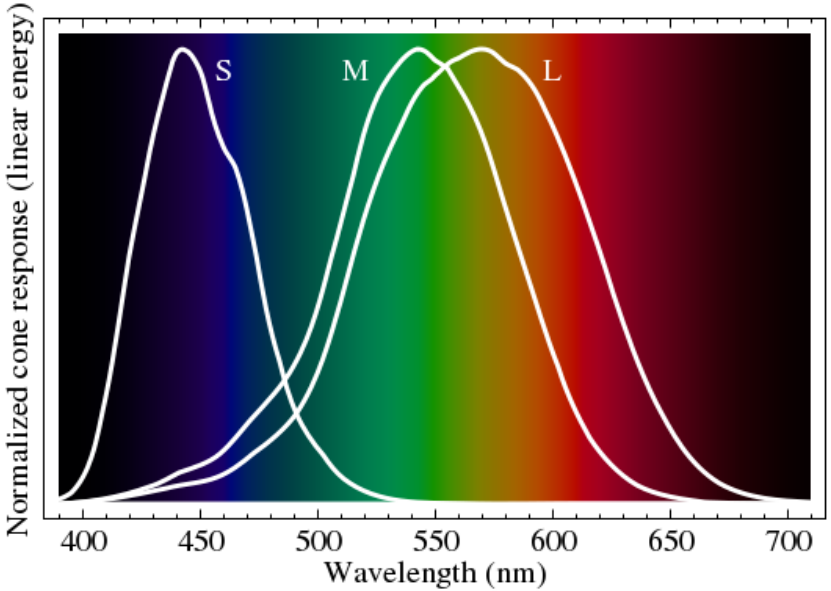
Cones (*pol. czopki*) color recognition:

- Can recognize 10 mln of different colors
- Three types of cones are sensitive to different wavelengths

Cone type	Name	Range	Peak wavelength
S	β	400–500 nm	420–440 nm
M	γ	450–630 nm	534–555 nm
L	ρ	500–700 nm	564–580 nm

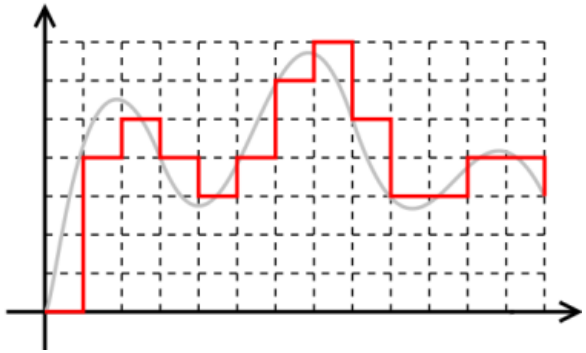
Cones (*pol. czopki*) color recognition:

- Can recognize 10 mln of different colors
- Three types of cones are sensitive to different wavelengths



Color quantization

- Color quantization is a transformation, from analog to digital representation of value.
- Light intensity represented by normalized range [0.0 ... 1.0] can be quantized to [0 ... 255] range.
- After quantization to such range, information about intensity can be stored using 8 bits, 1 byte.



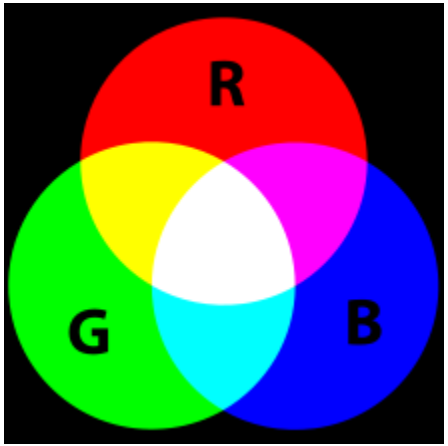
Color quantization

- Color quantization is a transformation, from analog to digital representation of value.
- Light intensity represented by normalized range [0.0 ... 1.0] can be quantized to [0 ... 255] range.
- After quantization to such range, information about intensity can be stored using 8 bits, 1 byte.

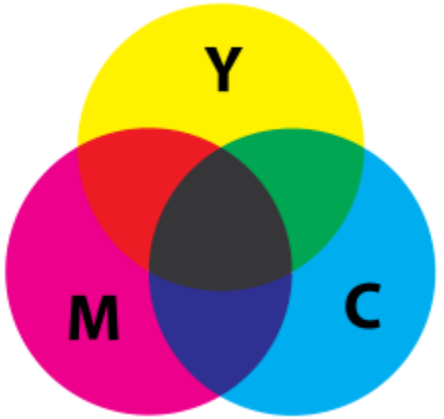
8bpp image of Lena
(monochrome, only intensity)

bpp (*bits per pixel*) – describes color depth, quality of image quantization.

Game Consoles and Hardware



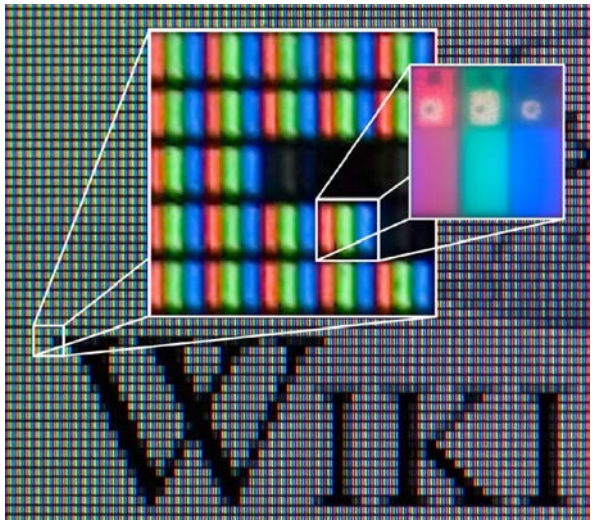
Additive models
(used in displays)



Subractive models
(used in poligraphy)

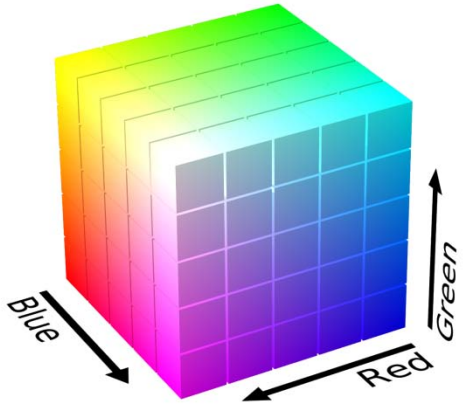
Additive color models

- Based on physical properties of light
- Takes into notice construction of human eye
- We „add” base colors to obtain white
- Used in all sorts of displays (CRT’s, LCD’s, projectors, etc.)



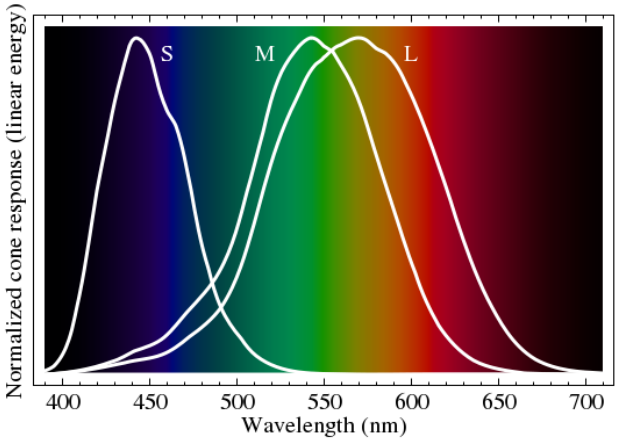
Most common additive models are RGB and sRGB

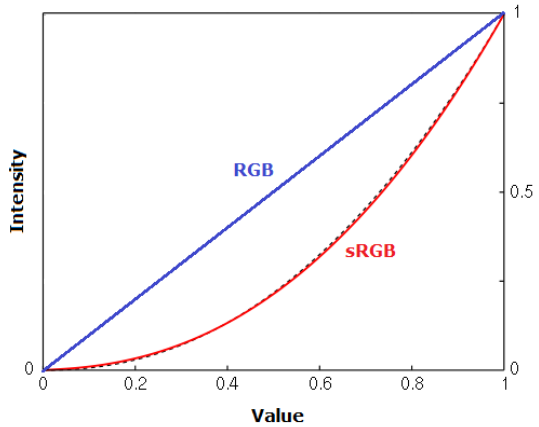
- Compilation of Red, Green and Blue base color intensities
- Intensities are normalized to [0.0 ... 1.0] range for LDR images, and can exceed 1.0 value for HDR images
- Commonly [0.0 ... 1.0] range is quantized to [0 ... 255] range which corresponds to 8 bits, 1 byte per base color (also called 24bpp color, or True Color)



sRGB (*standarised Red, Green, Blue*)

- RGB stores intensity of base colors in linear scale
- Human eyes are more sensitive to shadows and cannot differentiate highlights
- Non-linear scale is required for maintain image quality, in which more bits are used for low intensity and less for high intensity.





Game Consoles and Hardware

Subtractive color models

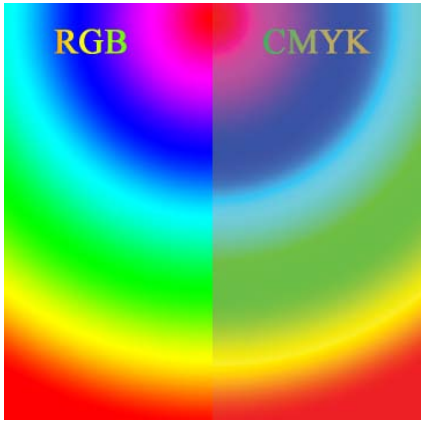
- Based on physical properties of pigments
- Used in polygraphy (banners, rollups etc.)
- We „subtract” colors to reach white

Most common model is CMYK:

- Cyan, Magenta, Yellow, Black
- White background as a base

Polygraphy basic rule:

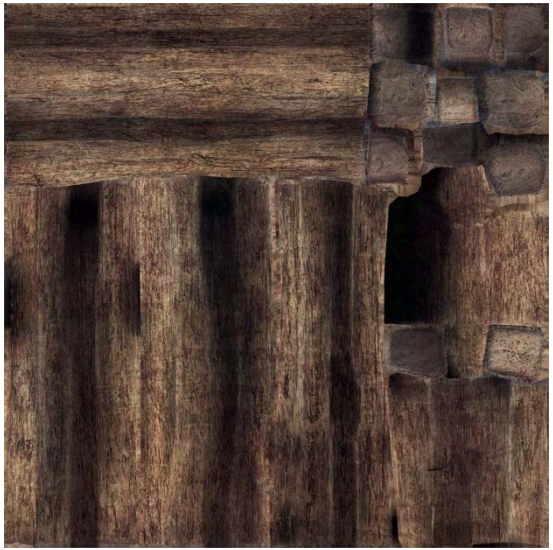
- Text and marks should be in vector representation, rest in raster (200dpi)



Difference between colors representation on screen (using RGB) and on paper (using CMYK).

TGA (*Truevision Graphics Adapter*)

- Can store alpha channel for transparency.
- Used by graphics as intermediate file format.
- It is also used in many games for storing final 2D assets of such elements like GUI or screenshots.
- Stores data uncompressed or compressed in lossless manner with RLE.

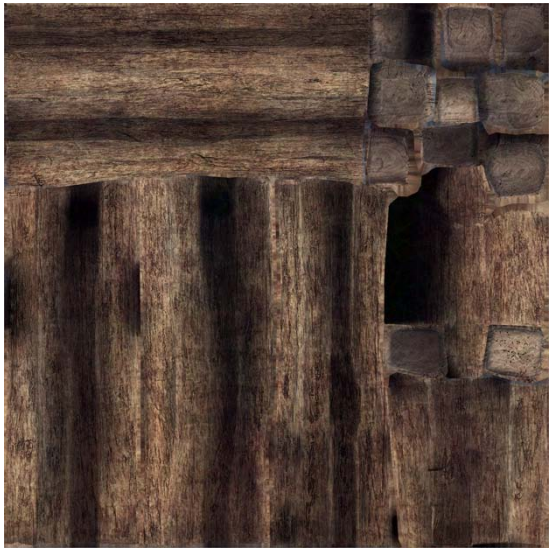


1024x1024 TGA texture

Lossless compression

(*pol. kompresja bezstratna*)

- Most common compression algorithms are LZ78 also know as RLE (*Run Length Encoding*) and LZ77.
- LZ77 is used in DEFLATE algorithm which is part of industry standard zlib compression library.
- Preferred way of storing images, if there is provided enough storage memory.
- Data compressed with RLE can sometimes take more space than uncompressed!

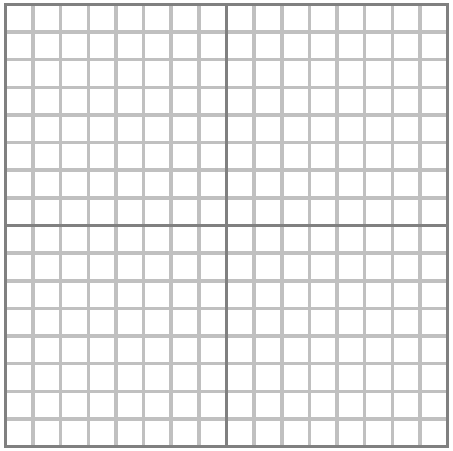


1024x1024 TGA texture

Format	Size in bytes
Uncompressed	3 145 746
RLE Compressed	3 152 629

PNG (*Portable Network Graphics*)

- Designed for internet use.
- Uses lossless compression LZ77.
- Can store alpha channel for transparency.
- Supports „Adam7” interlacing algorithm which allows displaying image even when having only partial data about it.
- Came out to be great format for games as well thanks to it’s good lossless compression and simplicity.



An illustration of Adam7 interlacing over a 16×16 image.

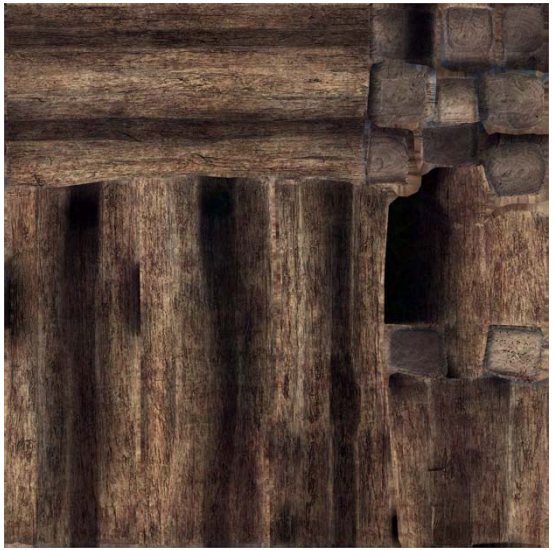
Lossy compression (*pol. kompresja stratna*)

- Used all types of advanced compression algorithms.
- Too complex for HW, needs to be decompressed in RAM to „BMP like” form before sending to graphic card. Good only for minimizing storage space.

DCT (*Discrete Cosine Transform*)

- Gives best results in terms of compression to quality ratio.
- Used in JPG but also in all sorts of other file formats like MP3, MPEG etc.
- Ideal in case of extremely limited storage space.

Comparison with compressed formats



1024x1024 TGA texture

Format	Size in Bytes
BMP	3 145 782
TGA Uncompressed	3 145 746
PNG at 6 in range 0-9	2 106 199
JPG at 66% ratio	179 263

Hint: You can use PNGCrush tool for minimizing compression size.

Lossy compression supported by HW

- Compression formats designed in the way they can be decompressed in flight by graphic hardware.
- Textures are stored compressed in both, storage and VRAM.
- Each HW vendor supports different compression algorithms.
- Image is compressed in fixed size blocks that can be accessed by GPU randomly (mostly 4x4 pixels).
- New ASCT algorithm supports variety of texel block sizes: 4x4, 5x4, 5x5, 6x5, 6x6, 8x5 ... up to 12x12

Lossy compression for HW comparison:

Name	DX10 Name	Description	Alpha Pre multiplied	Compression ratio
DXT1	BC1	Opaque (1bit Alpha possible)	N/A	6:1 for 24bpp
DXT2	(none)	Explicit Sharp Alpha	Yes	4:1
DXT3	BC2	Explicit Sharp Alpha	No	4:1
DXT4	(none)	Interpolated Gradient Alpha	Yes	4:1
DXT5	BC3	Interpolated Gradient Alpha	No	4:1
ETC1	(none)	Opaque (1bit Alpha possible)	N/A	6:1 for 24bpp
ASTC	(none)	2D, 3D, LDR, HDR, Alpha	No	4:1 up to 36:1
PVRTC	(none)	iOS Apple compression	Yes	6:1 up to 16:1

Game Consoles and Hardware

Currently available devices on the market:

Kinect for XBox 360

Creative Senz3D

Real Sense 3D*

Kinect v2.0 for XBox ONE

MYO Armband*

Game Consoles and Hardware

Depth Sensors capabilities

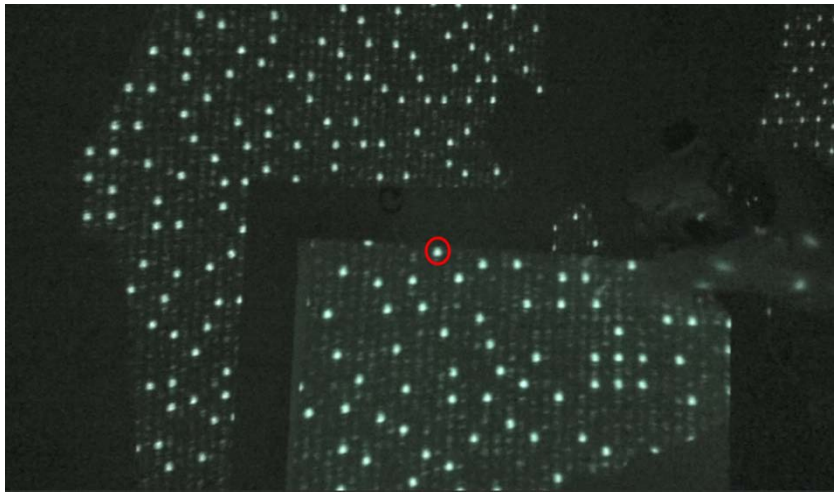
	Kinect for XBOX 360	Creative Senz3D	Kinect 2.0 for XBOX One	Real Sense
Sensor Type	Pattern	TOF	TOF	TOF (?)
Resolution	320x240	320x240	512x424	640x480
Field of View	H57.5 V43.5	H57.5 V43.5	H70.0 V60.0	(?)
Refresh Rate	30fps	60fps	30fps (?)	60fps (?)
Range	0.8m – 4.0m (0.4m-3.6m)	0.15m-1.0m	0.8m – 4.0m	0.15m-1.0m (?)
Accuracy	~1.5mm	1mm	~1.5mm	1mm (?)
Streams	Color, Depth	Color, Depth, IR, Vector Field	Color, Depth, IR	Color, Depth, IR, Vector Field
Color Resolution	640x480 (24bpp 30fps)	1280x720 (32bpp 30fps)	1920x1080 (16bpp 30fps)	1920x1080 (32bpp 30fps)

Game Consoles and Hardware

Kinect 360 (as well as first Kinect for Windows) has too low quality, and it's HW was designed for far range depth scanning (0.8m - 4.0m). It's also not portable and thus, not practical as integrated sensor.

It uses IR patterns to calculate scene depth as well as to recognise depth discontinuities.

It's real resolution is about 80 x 60 points which is used as a base for interpolated depth of 320x240.



Game Consoles and Hardware

Kinect for Xbox ONE (also known as Kinect v2.0 for Windows) has different depth sensor, it uses Time Of Flight depth camera which samples depth probing space one probe at a time.

This means that all depth probes in 514x424 resolution depth buffer provides meaningful data. It also has FullHD color camera which provides much more color samples to pick from.

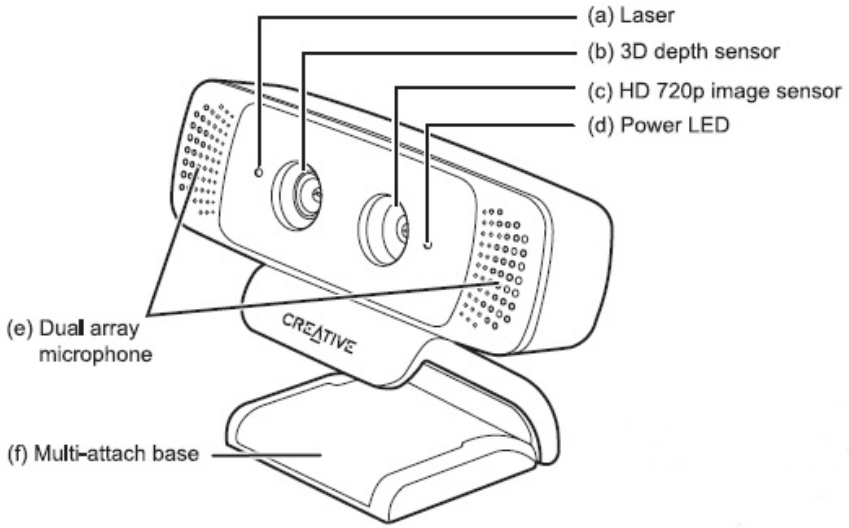
Unfortunately it’s size allows it to be used only as static sensor.

Game Consoles and Hardware

Creative Senz 3D thanks to it’s small size can be used as portable depth camera. It also uses TOF sensor but is designed for close range depth scanning and can provide frames with 60Hz frequency.

It’s a perfect fit for HMD mounted hands scanner, and it also provides additional data like vector field buffer or ID buffer.

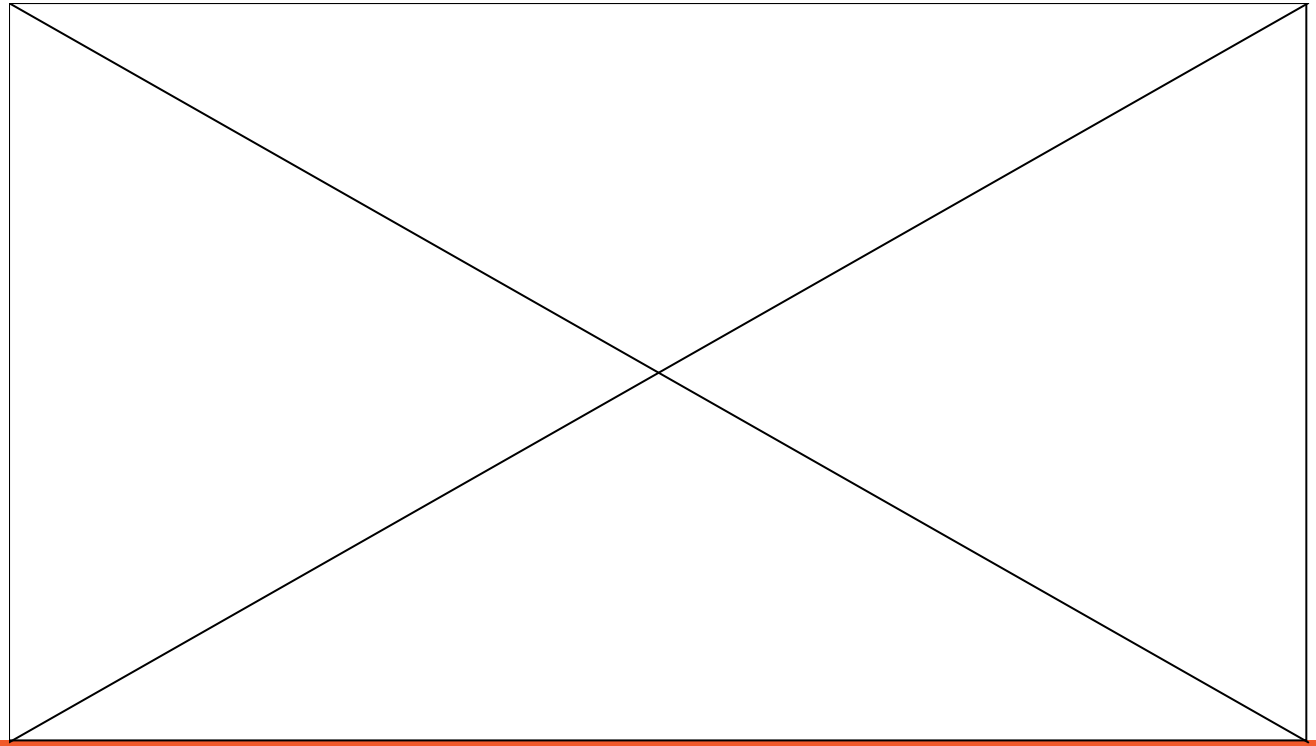
(Intel Perceptual Computing SDK that is used to access the data is somewhat non trivial to use :/)



Depth Sensor example usage in VR:

Game Consoles and Hardware

Wolfenstein VR:



Game Consoles and Hardware

Mobile market is evolving so fast, that it is already becoming similar to PC market in variety of different configurations, brands and HW capabilities. Here you can see Android Market share of different brands:

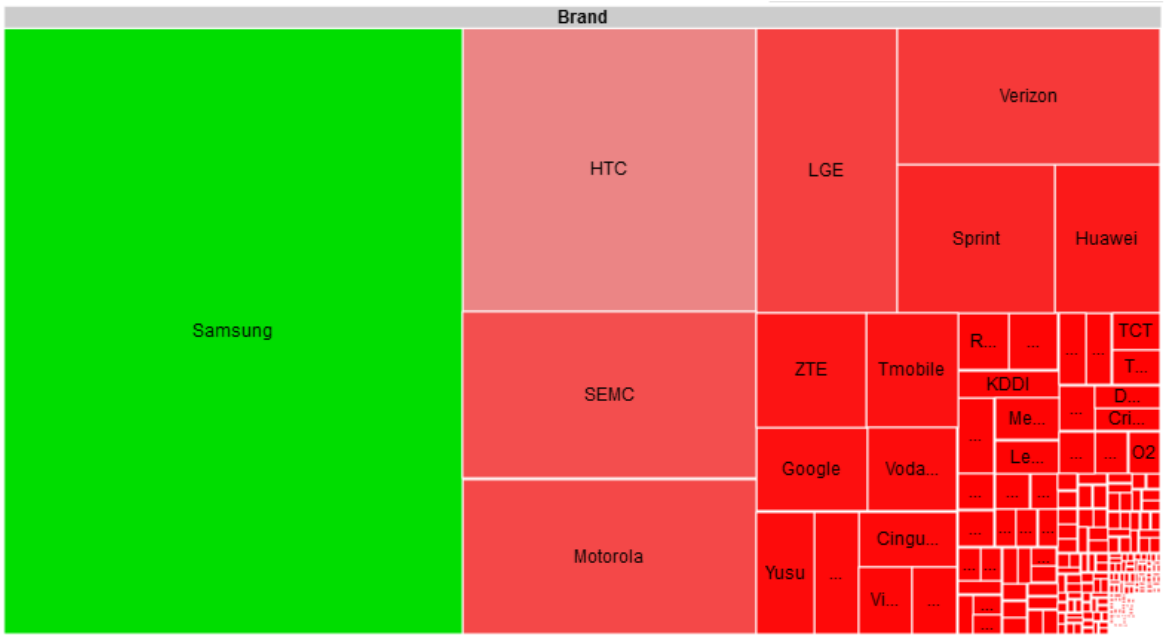


Image is courtesy of OpenSignalMaps

Game Consoles and Hardware

All this leads to huge fragmentation known from PC market. Below is shown popularity of different device models using Android:

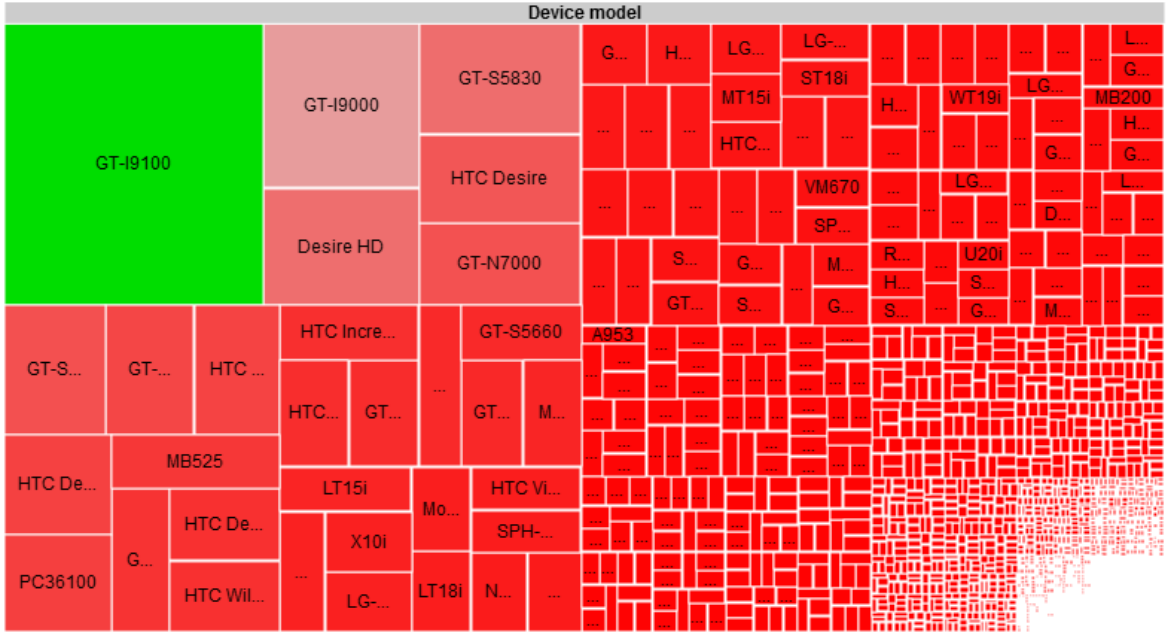


Image is courtesy of OpenSignalMaps

Game Consoles and Hardware

Check what devices do you have:

	Android	BlackBerry	iOS	Windows
Default IDE:	Eclipse	Momentics	XCode	Visual Studio
Default language:	Java	C++/AIR/HTML5	Objective-C	C/C++
Required devices:	- **Android 2.3+** - Phone or Tablet	- Emulator - **Dev Alpha** - PlayBook, **Z10**, Q10	- **MacBook** - Emulator - **iPhone**, iPad, iPod	**None**

What about Windows Phone?

Features available from version 8:

DirectX 9 like interface for rendering, SM 2.0 HLSL.

Native development in C++

WP is not worth multiplatform development effort at this moment.

Game Consoles and Hardware

Common development environment:

	Android	BlackBerry	iOS	Windows
Native development:	Yes	Yes	No	Yes
Language for games:	C++	C++	C++ (Objective-C++)	C/C++
Wrapper required:	Optional	No	Required	No
Debug build VS:	VS Android	Plugin	Only on PC	Yes
Release build VS:	VS Android (?)	Plugin	No	Yes
Debugging in VS:	Plugin console	Plugin	Only on PC	Yes
Graphics:	OpenGL ES	OpenGL ES	OpenGL ES	OpenGL
Audio:	OpenAL	OpenAL (OpenSL)	OpenAL	OpenAL

Game Engines in Poland

CI Games
People Can Fly
The Farm 51
Astronauts
Dark Stork
Nicolas Games

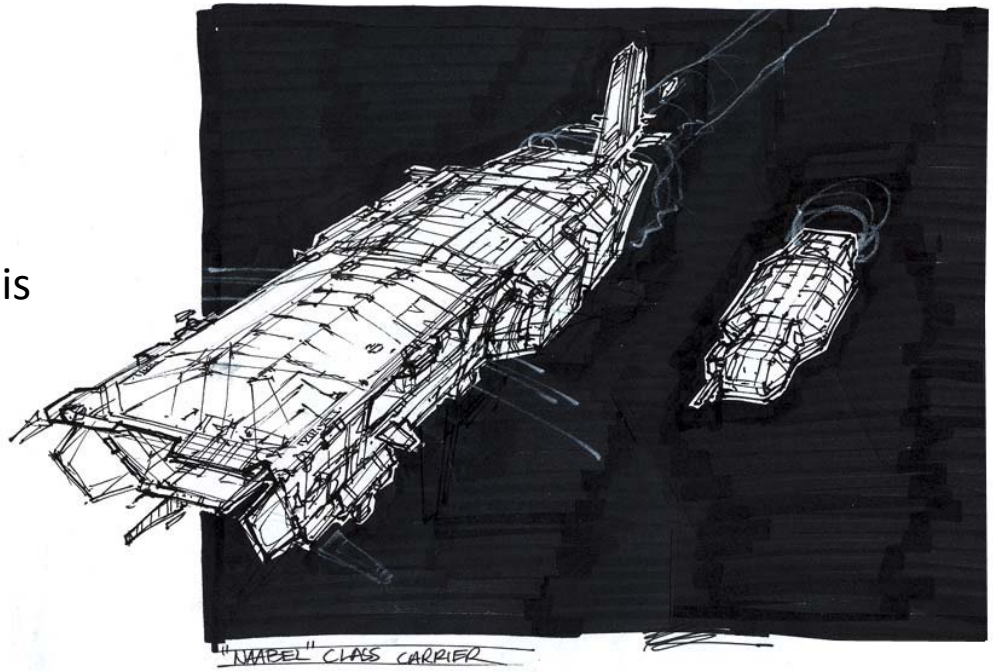
Fuero Games
Crunching Coalas
GameLab
Ganymede
Infinity Dreams
INTERMARUM
Jujubee
Tap It Games
AideMMediA
Anshar Studios
Nawia Games
…

CD Projekt RED
Techland
Reality Pump
Flying Wild Hog
Vivid Games
Artifex Mundi
11bit Studios
Teyon
Tate Interactive
QubicGames
PlayWay
Nitreal Games
Nano Games
Nano Titans
Gamelion
EXOR Studios
Big Daddy's Creations
Drago Entertainment
…

Asset Pipeline

Concept Art

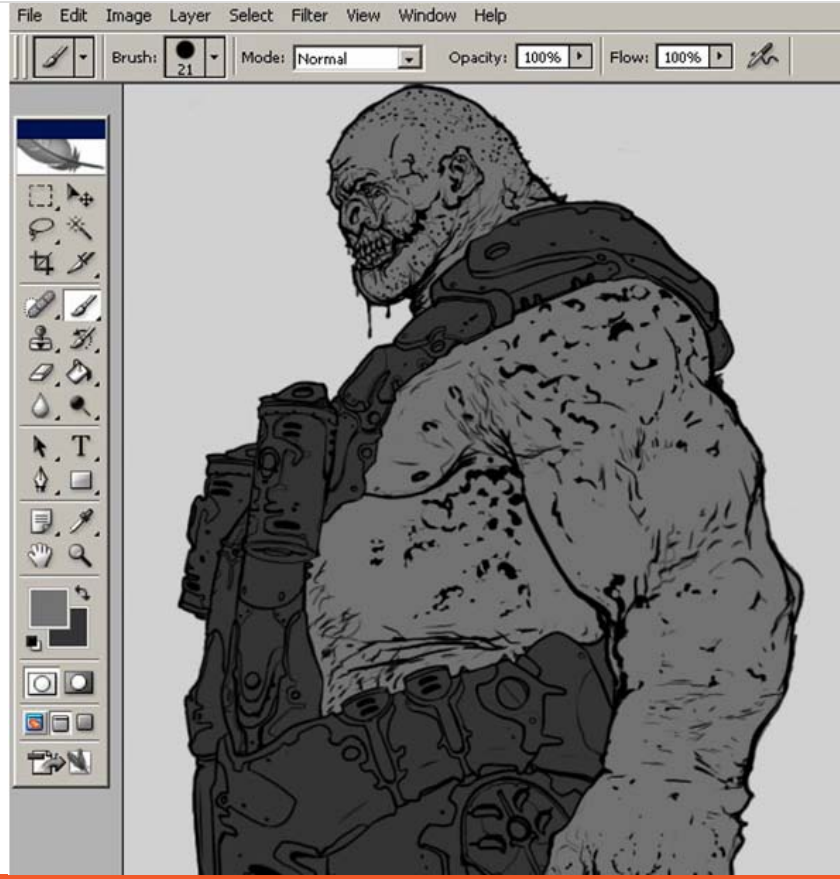
- To create initial design and shape, concept arts for few projections and situational drafts are made.
- For humans and creatures this is few poses in action.
- Traditional way was paper, pencils, markers



Homeworld Concept Art (~1997)

Concept Art

- Currently all content and prototyping is done in software which simplifies whole process.
- On right, concept art of “Rabagh The Bombardier” done by Vitaly Bulgarow in Photoshop.
- We will use his work as an example of next-gen asset creation.
- Whole tutorial can be bought here.



Game Dev School

Asset Pipeline

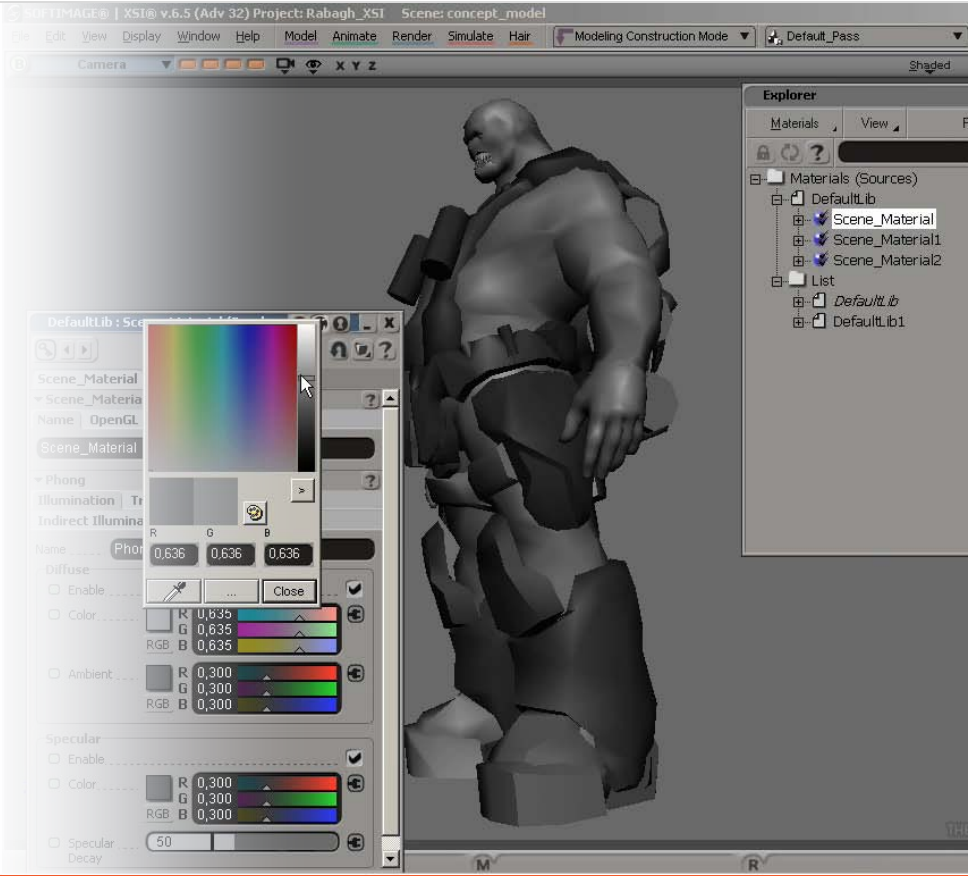
Concept Art

- Final concept art is ready. Now it is time to create model based on it.
- Concept arts can be used directly during modeling as a background for reference.
- They can be also set as a three projections for each of the axes, which is useful during creation of vehicles and buildings.

Asset Pipeline

Modeling

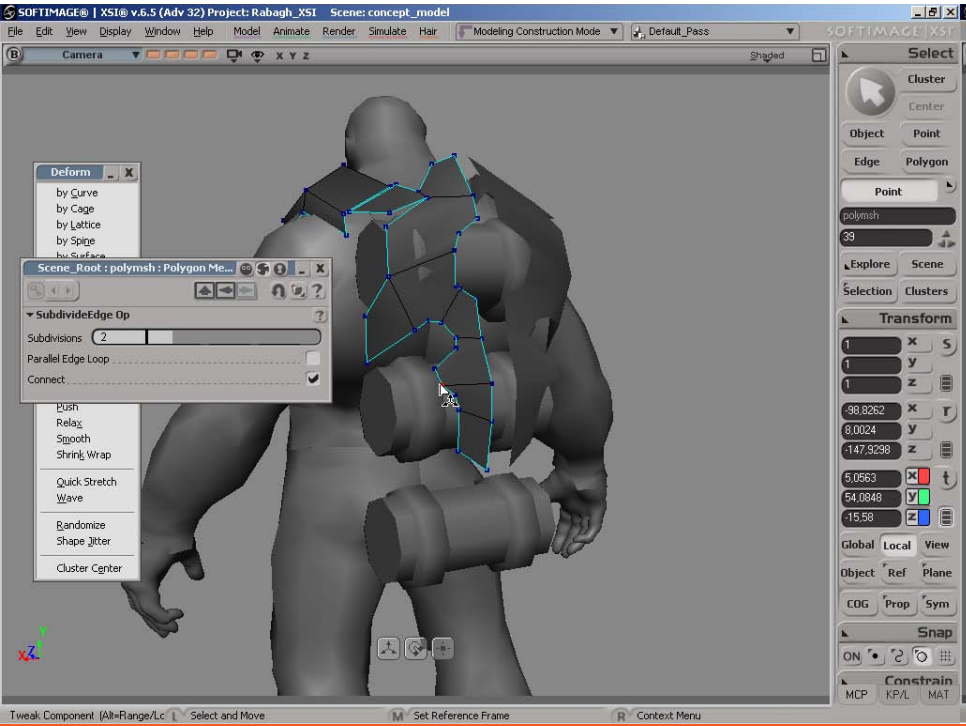
- Base human model with animation skeleton is used as a starting point.
- It is divided into parts that will have different materials.
- Each part is modeled separately to get the final outcome.
- Here initial phase of “Rabagh” in Softimage XSI 6.5

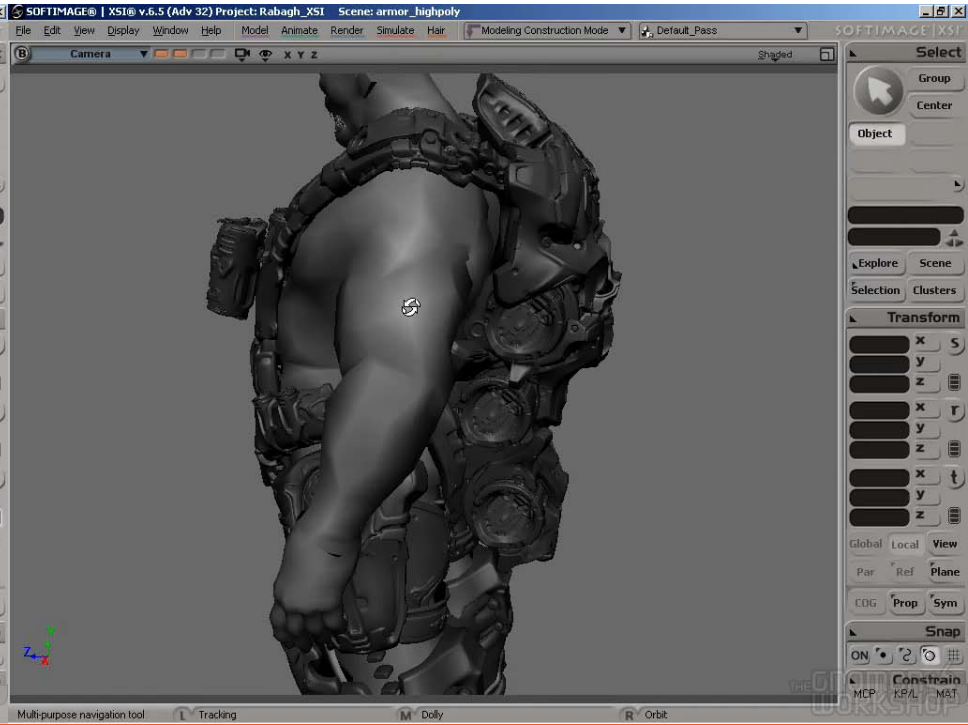


Asset Pipeline

Modeling

More popular in Japan than in USA. In Europe used for example by Lionhead Studios and 11bit Studios. Good as an additional asset in portfolio.

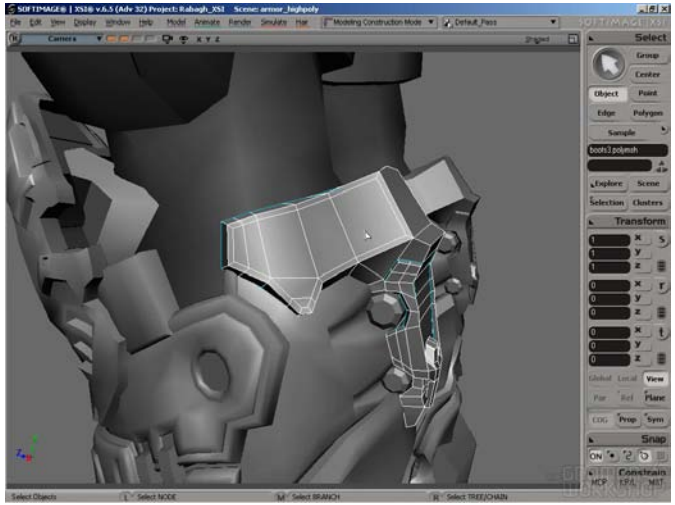




Asset Pipeline

Modeling

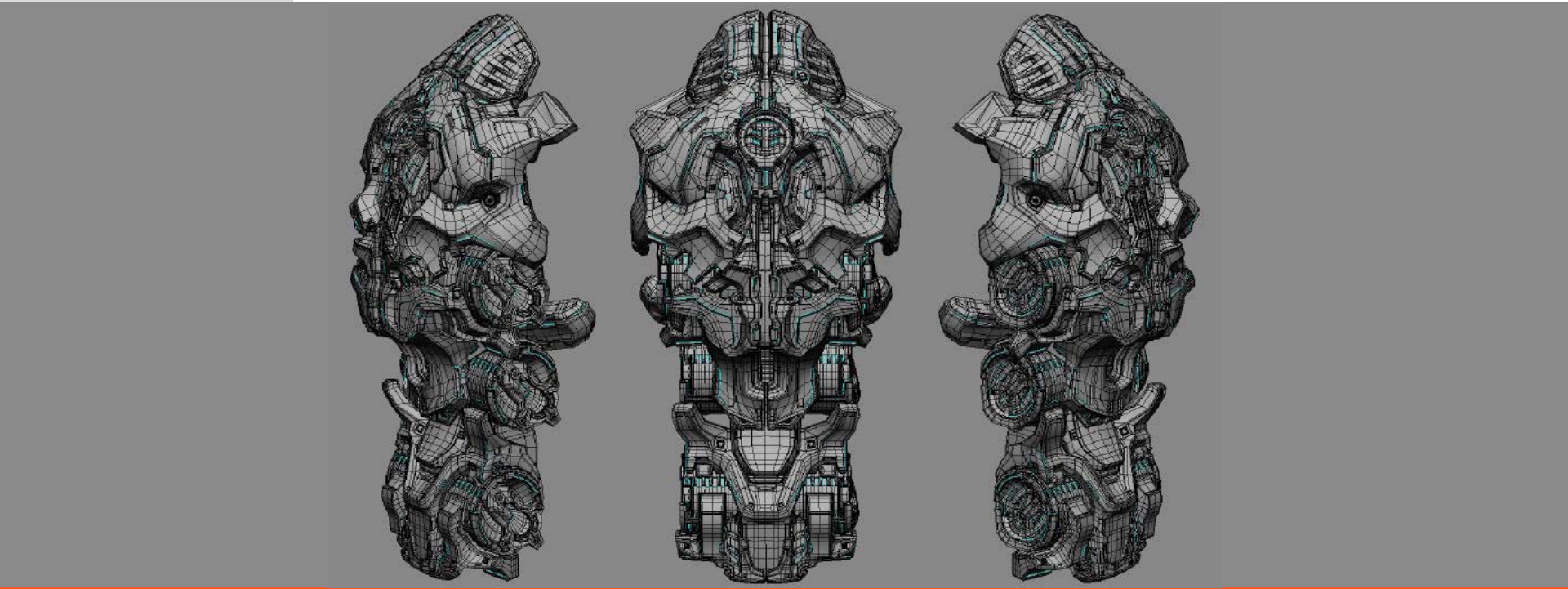
Iterating step by step from low poly, high poly model is created with all the final detail.



Asset Pipeline

Modeling

Technical parts with hard surfaces like this backpack were modeled in Softimage. Here in wireframe mode.



Asset Pipeline

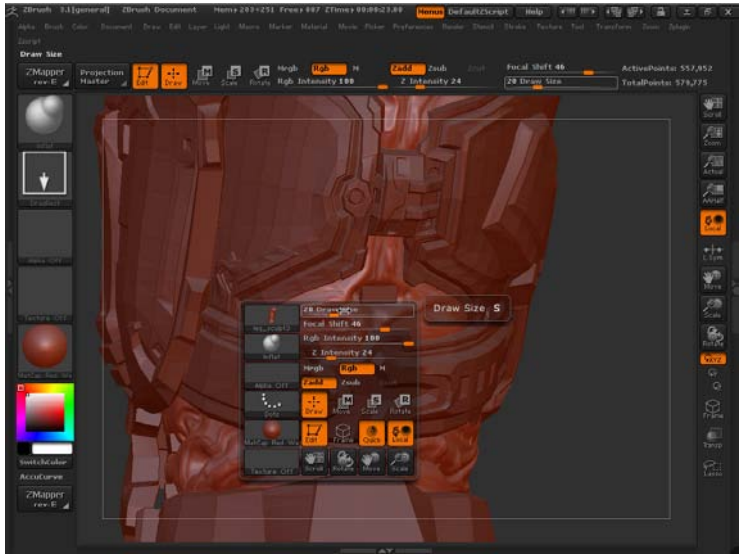
Modeling

Final high poly model of backpack presented using offline rendering.

Asset Pipeline

Modeling

Sculpting tool like ZBrush is ideal for modeling high-poly organic parts (it's like forming clay).



Asset Pipeline and Creation

Asset Pipeline

Modeling

High-poly model is ready.

It is build from few millions of polygons.

It is definitely too much geometry for any graphic card to render in real time.

Based on high-poly version, low-poly model is generated. It can be done automatically by tools, and then corrected manually by the artist.

Here you can see 35K poly version of model.

It will be used to render enemies located very close to the player.

Asset Pipeline

Modeling

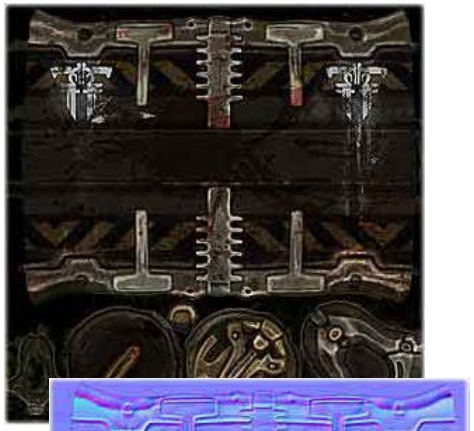
When required polygon budget is reached during generation of low-poly model, both versions are ready. Now it is time to create textures.

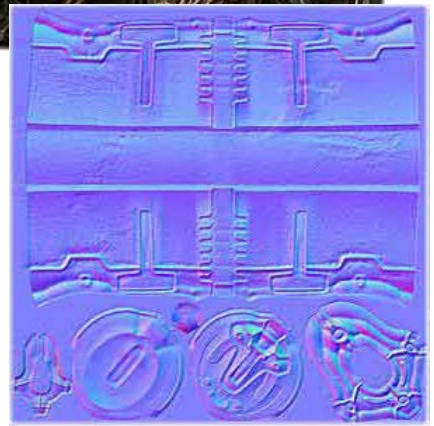
Texturing

- Each low-poly part of model is now sliced and projected on 2D surface.
- As a result each point of model gets coordinates in 2D space of texture.
- This is called unwrapping and allows creation of textures that will be used with model.
- Here you see three 2048x2048 already painted textures for legs, belt and organic surfaces.

Texturing

- Using difference in shape between high-poly and low-poly model, it is possible to automatically generate displacement maps (called also height maps).
- From them also normal maps are generated.
- Displacement, height and normal maps will be explained further in this presentation.
- Two 1024x1024 textures with their normal maps.





Asset Pipeline

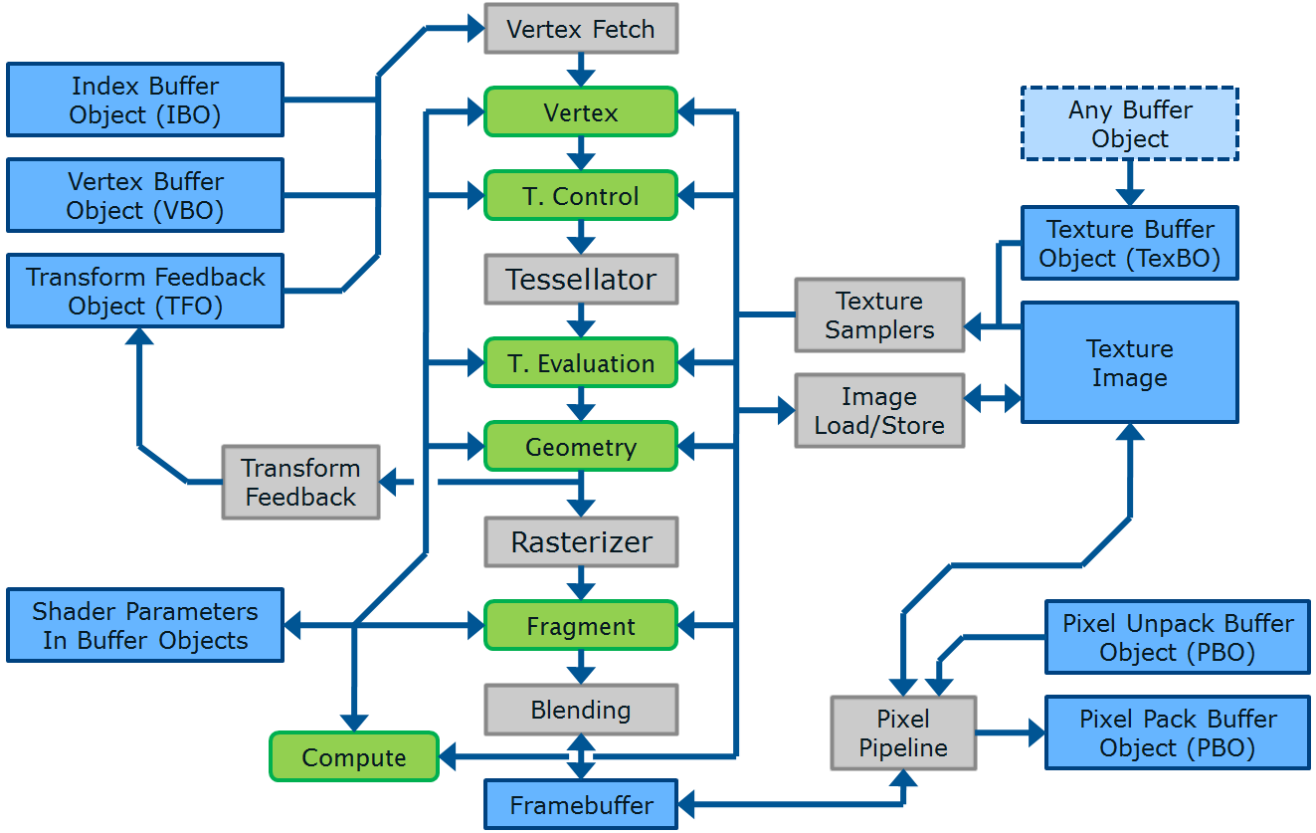
Final model with full set of textures is ready. To be used in game it still needs whole set of different animations to be created like walking, crouching, jumping etc. Technical Artist will also need to create final materials for it, using prepared diffuse, specular, normal and displacement textures.

GPU Architecture - Rendering Pipeline

Rendering Pipeline

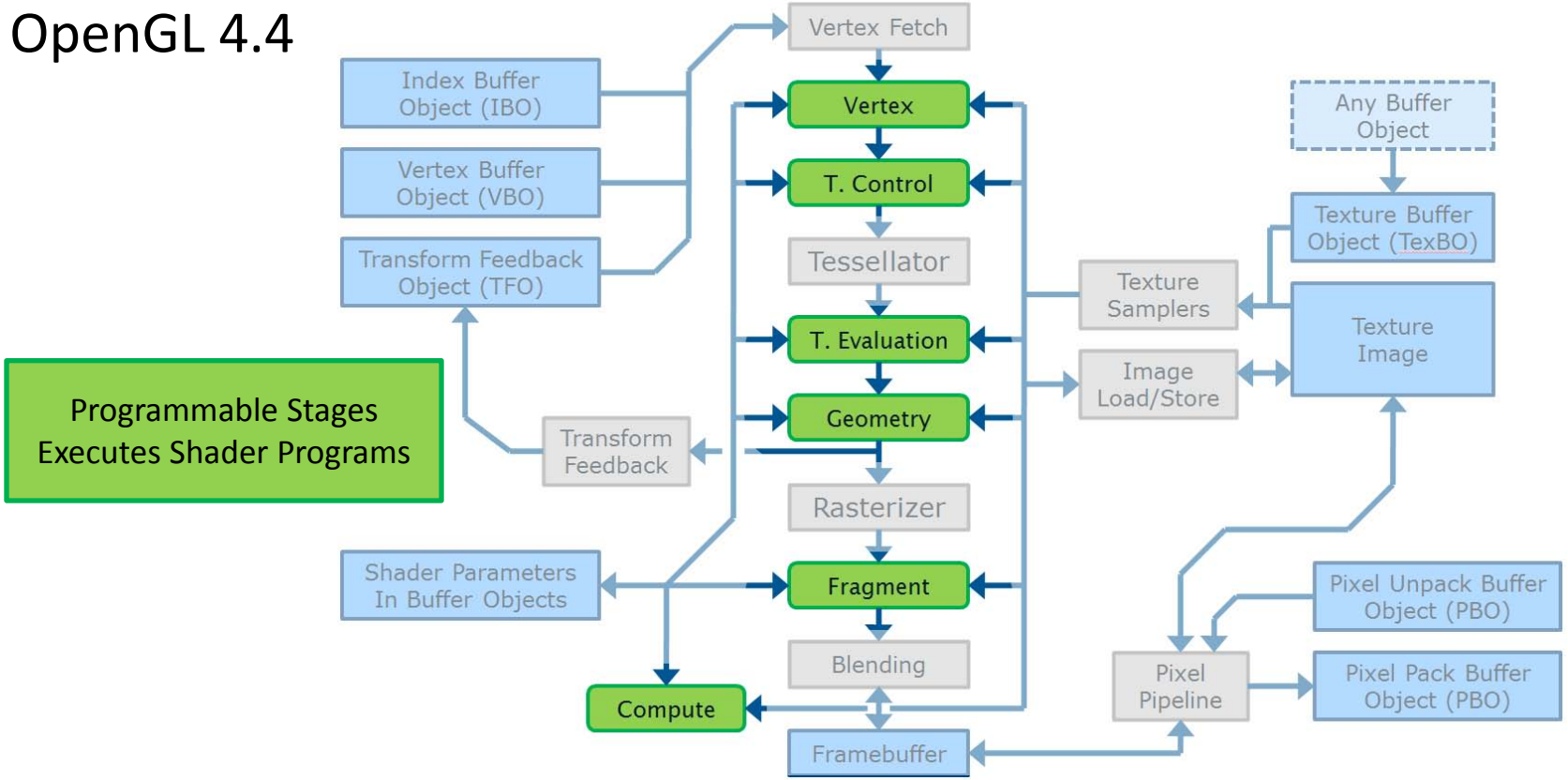
GPU Architecture - Rendering Pipeline

OpenGL 4.4



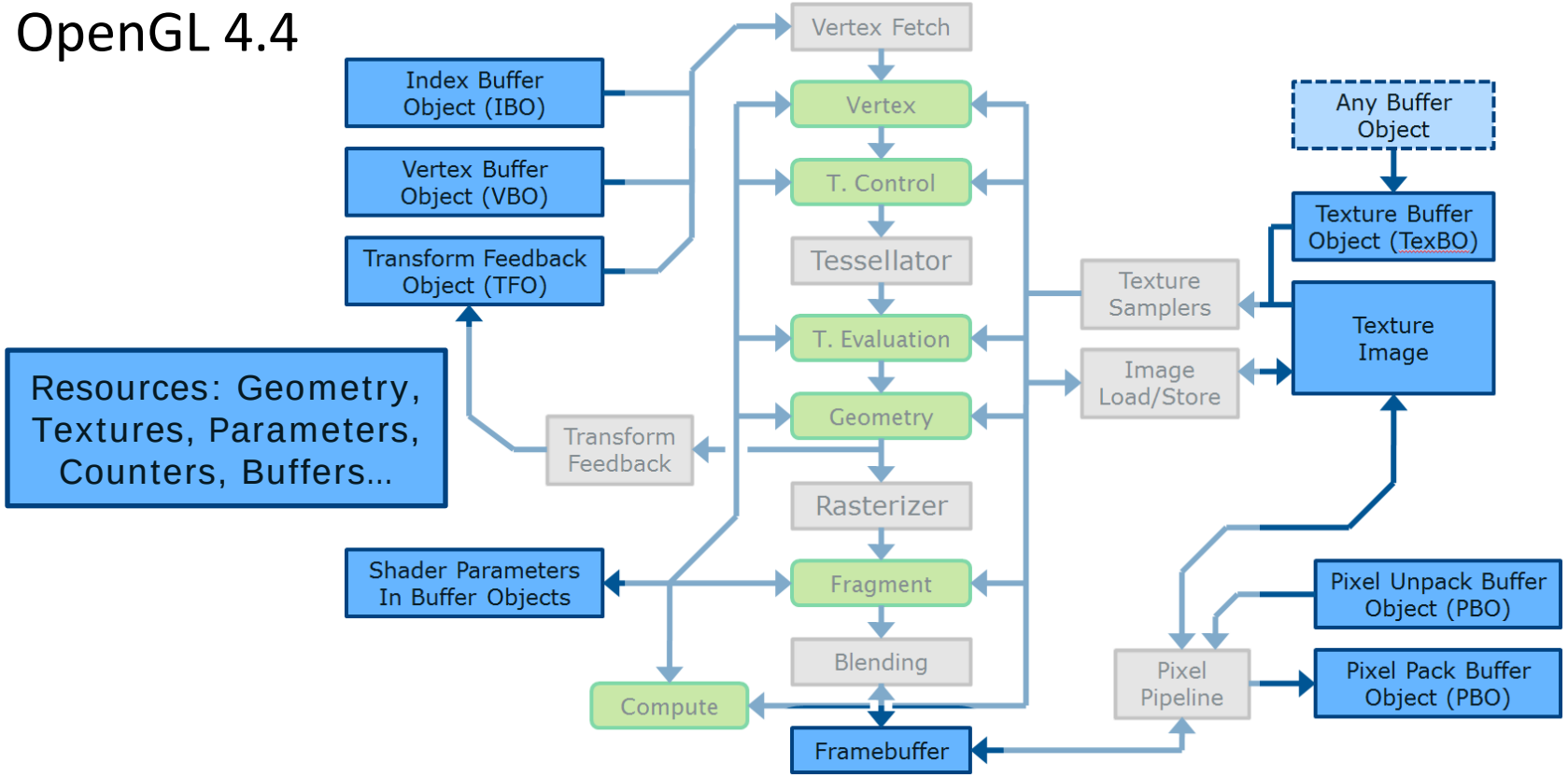
GPU Architecture - Rendering Pipeline

OpenGL 4.4



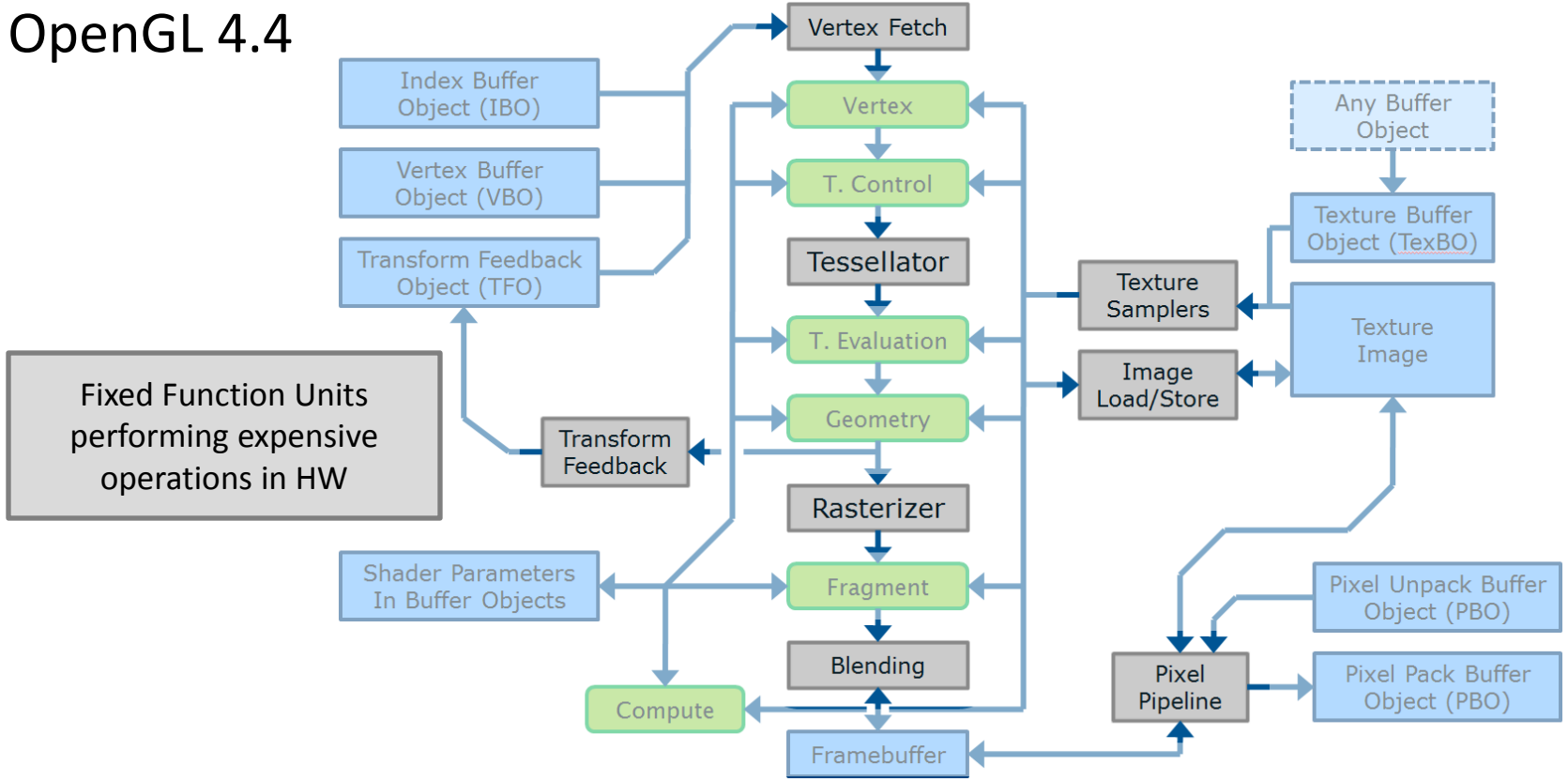
GPU Architecture - Rendering Pipeline

OpenGL 4.4



GPU Architecture - Rendering Pipeline

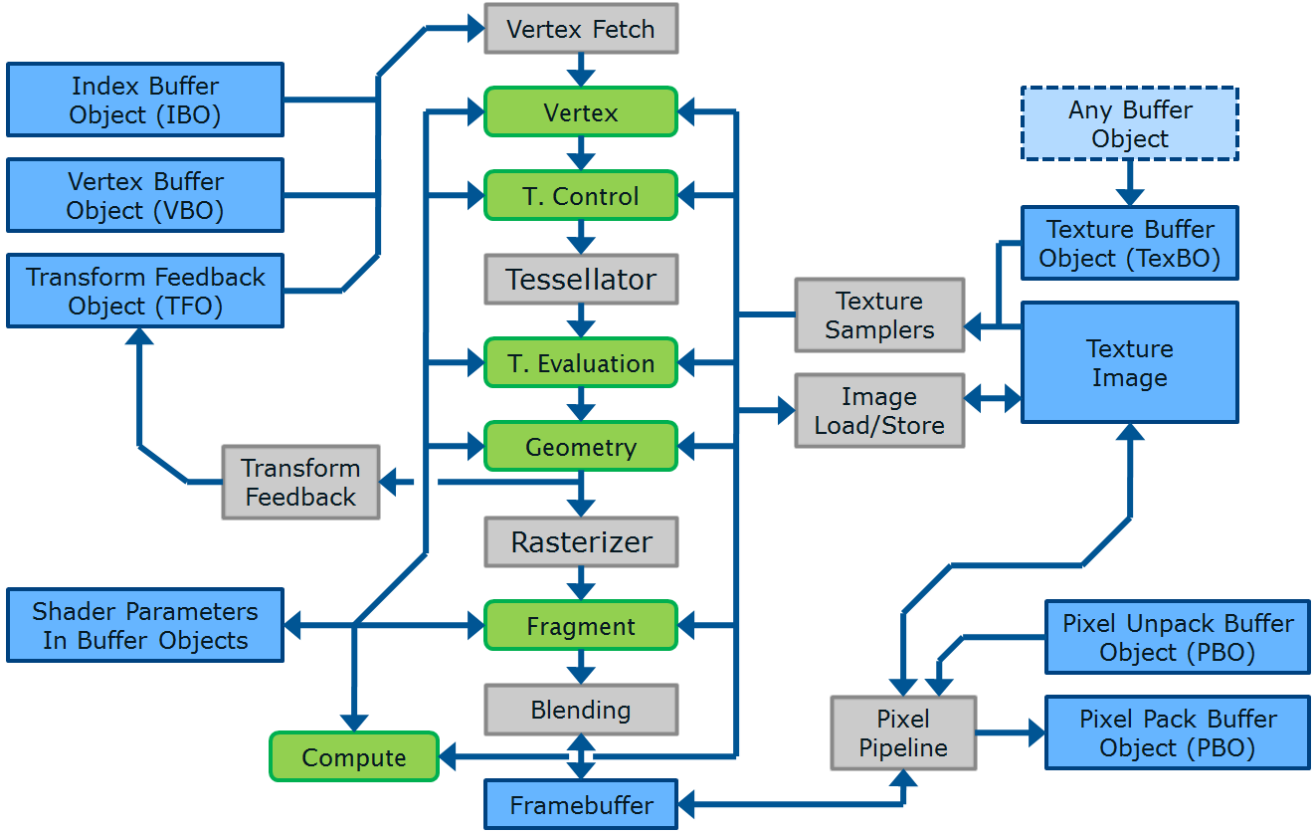
OpenGL 4.4



Vertex Shader

GPU Architecture - Rendering Pipeline

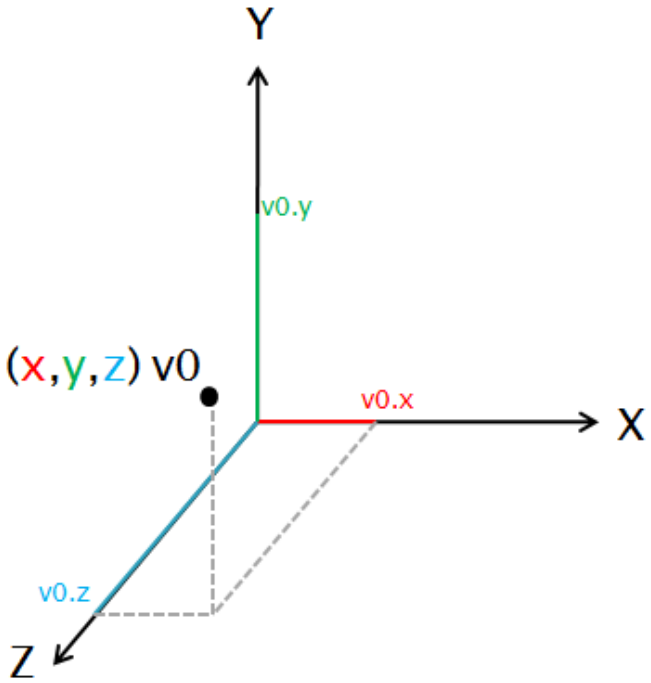
OpenGL 4.4



GPU Architecture - Rendering Pipeline

Vertex (*pol. wierzchołek*)

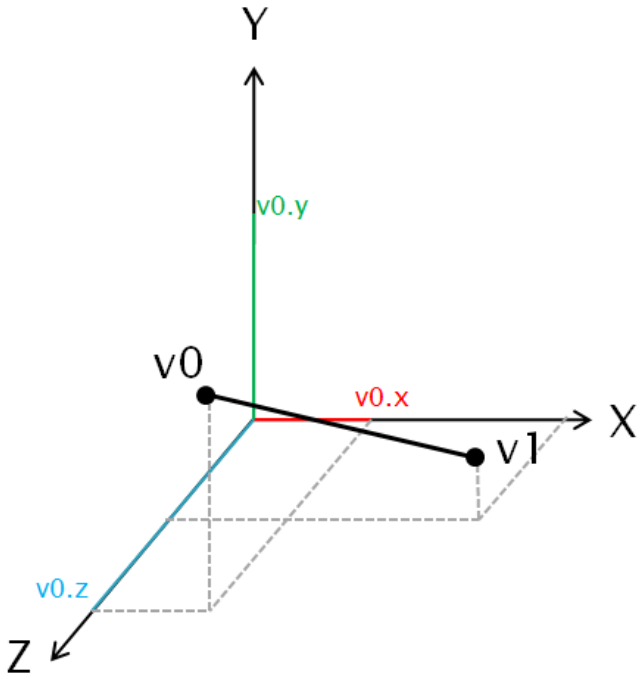
- has position in 3D space
- can have other attributes
  - color, texture coordinates, animation weights, etc.
- is a base primitive used to describe all other types of geometry primitives that can be processed by graphic cards



GPU Architecture - Rendering Pipeline

Line (*pol. linia*)

- build from two vertexes
- mostly used in specialized CAD software for displaying wireframes of designs
- in games used for rendering hairs, vector graphics in very old games, etc.



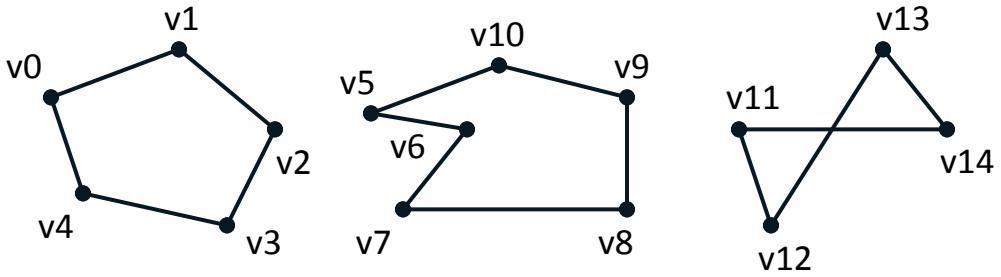
GPU Architecture - Rendering Pipeline

Vertex and Line types accepted by OpenGL:

Points
(GL_POINTS)

Separate Lines
(GL_LINES)

Line Loops
(GL_LINE_LOOP)



GPU Architecture - Rendering Pipeline

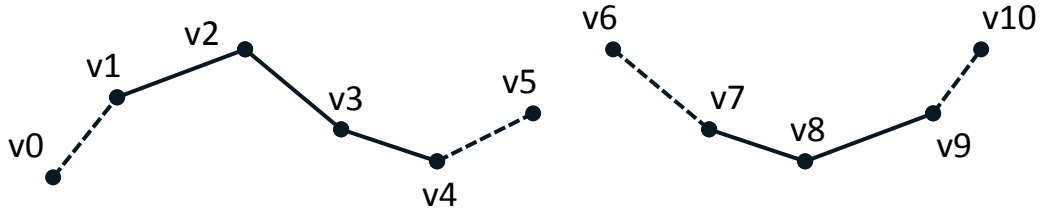
Vertex and Line types accepted by OpenGL:

Line Stripes
(GL_LINE_STRIP)

Special Types*:

Separate Lines
with Adjacency
(GL_LINES_ADJACENCY)

Line Stripes
with Adjacency
(GL_LINE_STRIP_ADJACENCY)

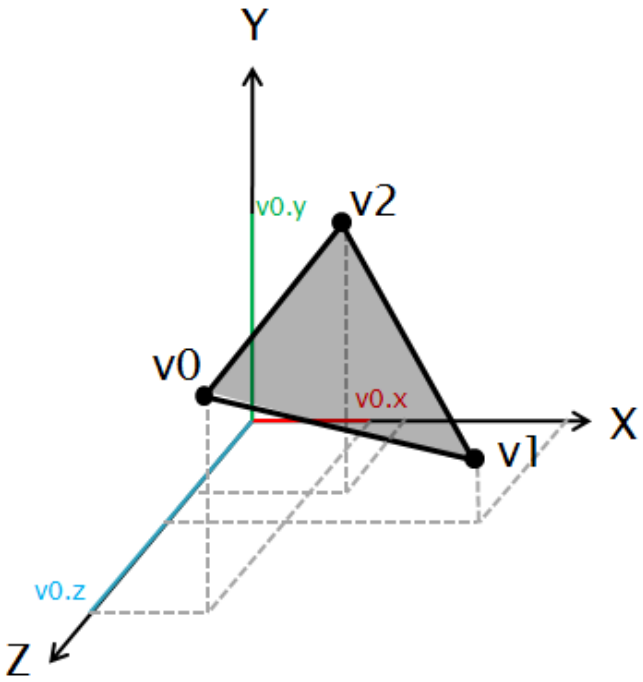


*Will be explained with Geometry Shaders.

GPU Architecture - Rendering Pipeline

Triangle

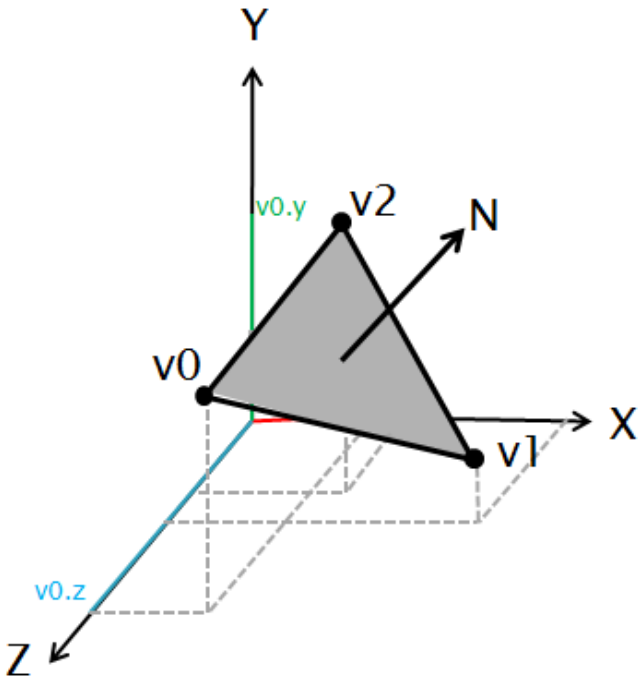
- simplest primitive capable of describing a surface
- most important primitive in computer graphics
- all three dimensional forms are build from triangles
- used also in 2D graphics to describe User Interfaces, sprites, etc.



GPU Architecture - Rendering Pipeline

Normal vector

- vector perpendicular to triangle surface
- describes towards which direction triangle is facing, which side of it is visible



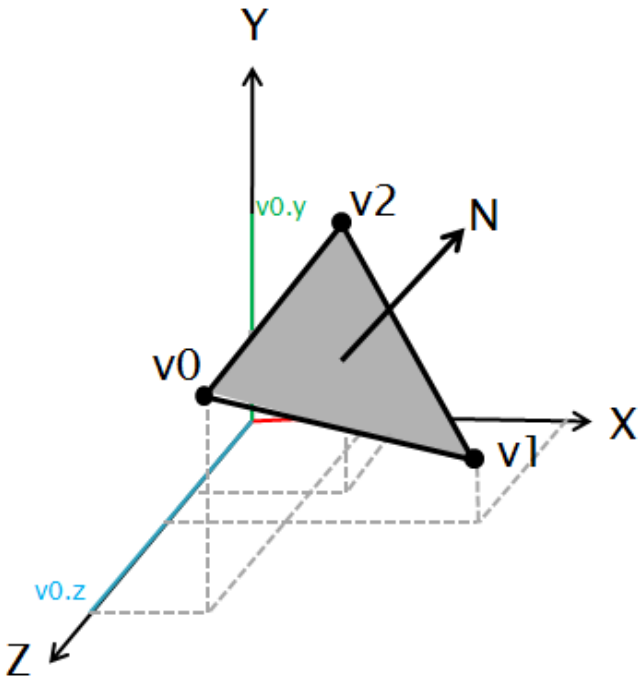
GPU Architecture - Rendering Pipeline

Normal vector

Used for:

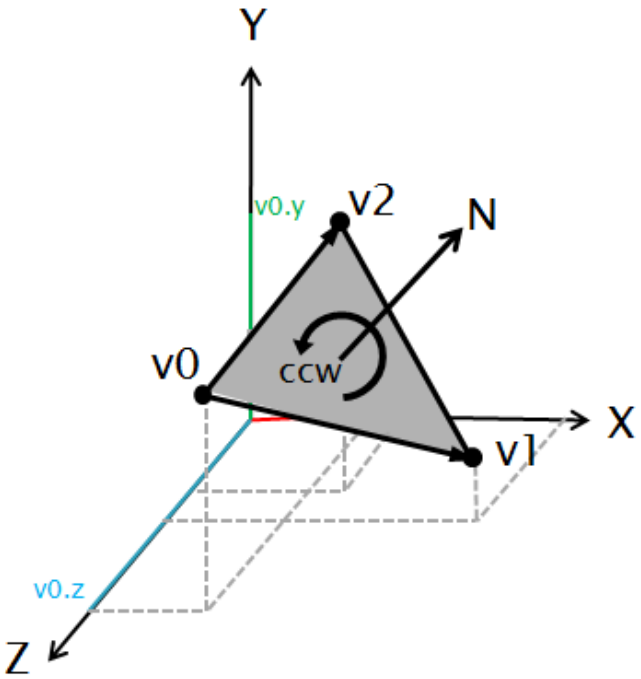
- visibility check (back face culling)
  each triangle has two sides: visible and invisible,
  when normal is pointing towards observer - it's visible
- surface lightning properties
  needed for light calculation - determine how much
  light is reflected from surface to observer
- tessellation calculations

  normal vectors are used during displacement of
  vertices generated by Tessellator
- distance calculation
  dot product of normal and point returns distance from
  surface defined by triangle to this point



Normal vector

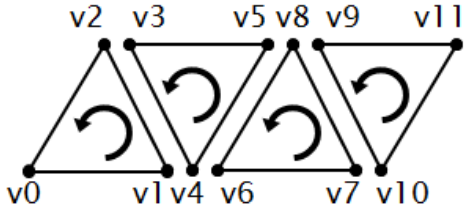
- calculated using cross product, possible two directions of calculation:
  - right hand rule
    CCW (Counter Clockwise)
    N = (v1 - v0) x (v2 – v0)
  - left hand rule
    CW (Clockwise)
    N’ = (v2 - v0) x (v1 – v0)
  - N = -N’



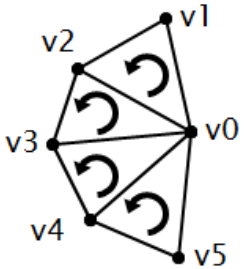
GPU Architecture - Rendering Pipeline

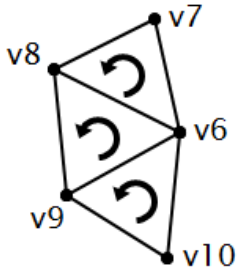
Triangle types accepted by OpenGL:

Separate Triangles
(GL_TRIANGLES)

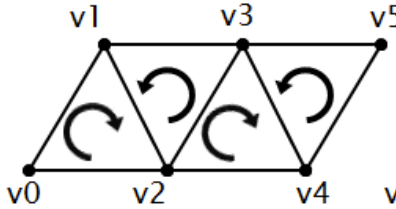


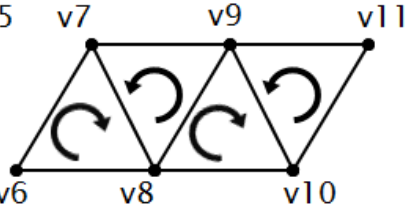
Triangle Fans
(GL_TRIANGLE_FAN)





Triangle Stripes
(GL_TRIANGLE_STRIP)

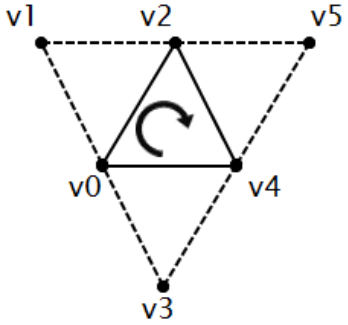




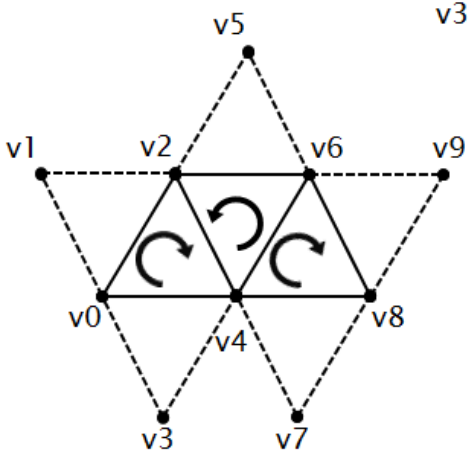
GPU Architecture - Rendering Pipeline

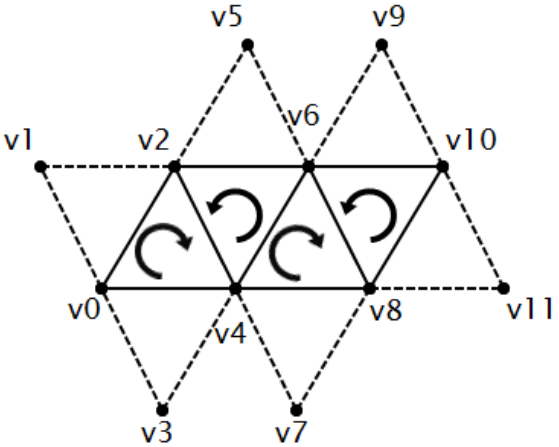
*Special Triangle types accepted by OpenGL:

Separate Triangles
with Adjacency
(GL_TRIANGLES_ADJACENCY)



Triangle Stripes
with Adjacency
(GL_TRIANGLE_STRIP_ADJACENCY)

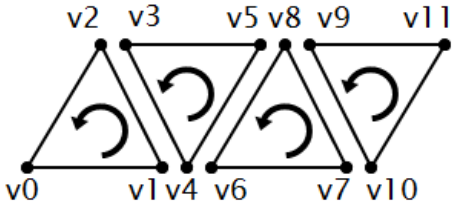




*Will be explained with Geometry Shaders.

GPU Architecture - Rendering Pipeline

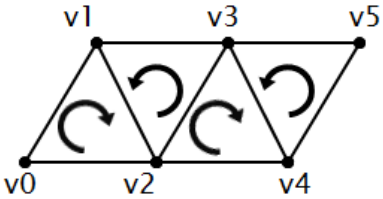
Separate Triangles



- Most 3D models are closed.
- This means each triangle has surrounding neighbor triangles with which it shares its vertexes.
- It would be a waste of VRAM, bandwidth and compute power to send the same vertex N times for each triangle that uses it.
- Specialized programs can generate triangle stripes.

VS

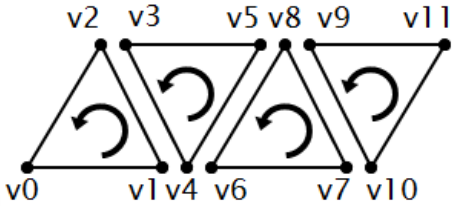
Triangle Stripes



- In Triangle Stripe each triangle is created from current vertex and previous two.
- This causes vertex order to be reversed every second triangle, which means their normal vectors would be wrongly calculated.
- During rendering GPU takes it into notice to preserve normal's.

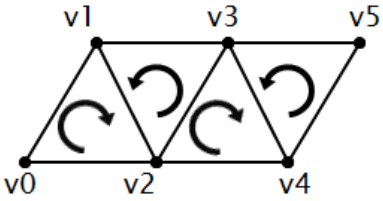
GPU Architecture - Rendering Pipeline

Separate Triangles



VS

Triangle Stripes



GPU Architecture - Rendering Pipeline

Explosions, gas, smoke

- Animation and movement of big amount of simple particles with given lifecycle and emitters

Foliage and plants

- Animation of grass, leaves and small plants based on force of wind and it's direction
- Can also take into notice for eg. Surrounding explosions

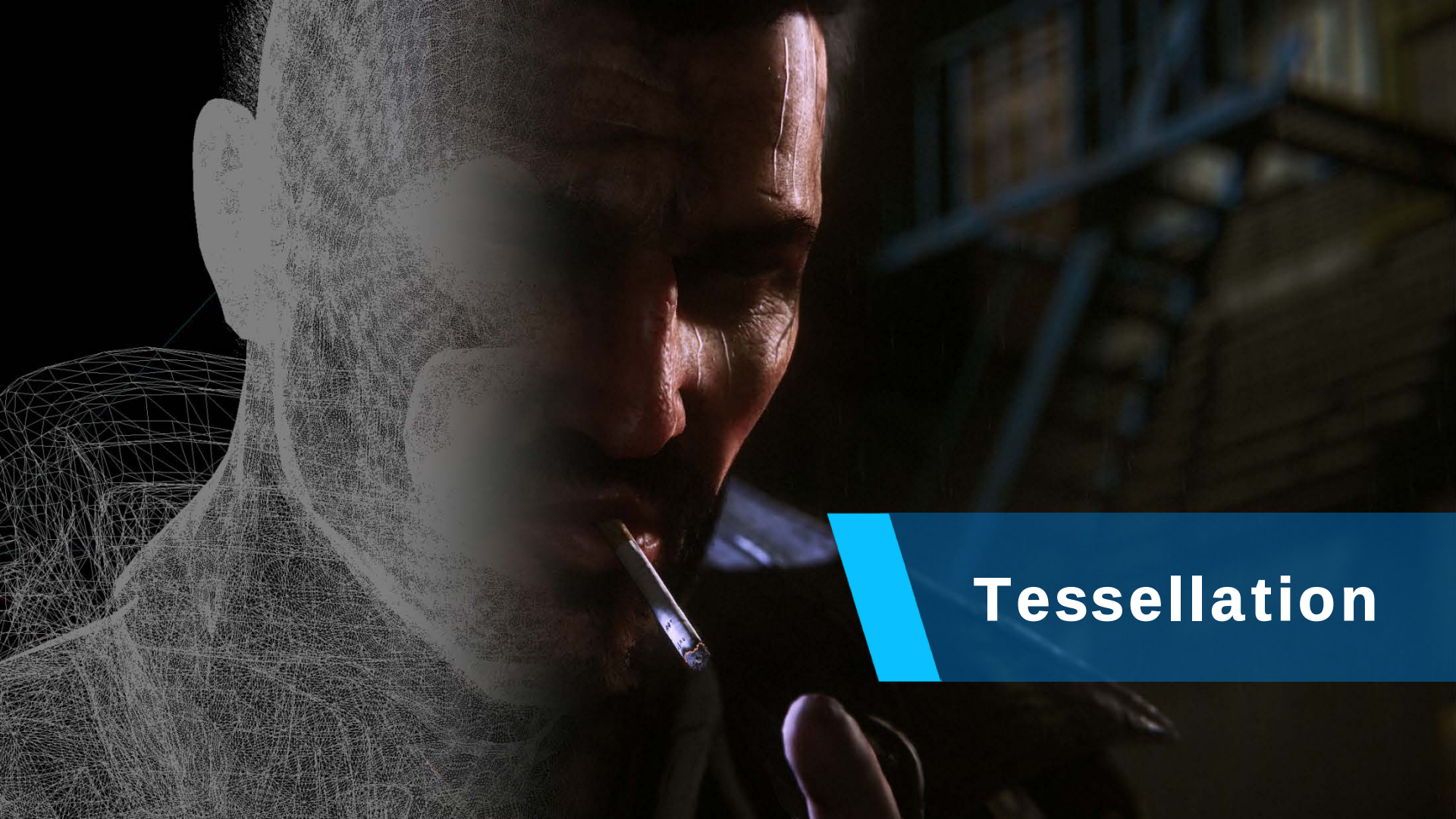
GPU Architecture - Rendering Pipeline

Explosions, gas, smoke

- Animation and movement of big amount of simple particles with given lifecycle and emitters

Foliage and plants

- Animation of grass, leaves and small plants based on force of wind and it's direction
- Can also take into notice for eg. Surrounding explosions

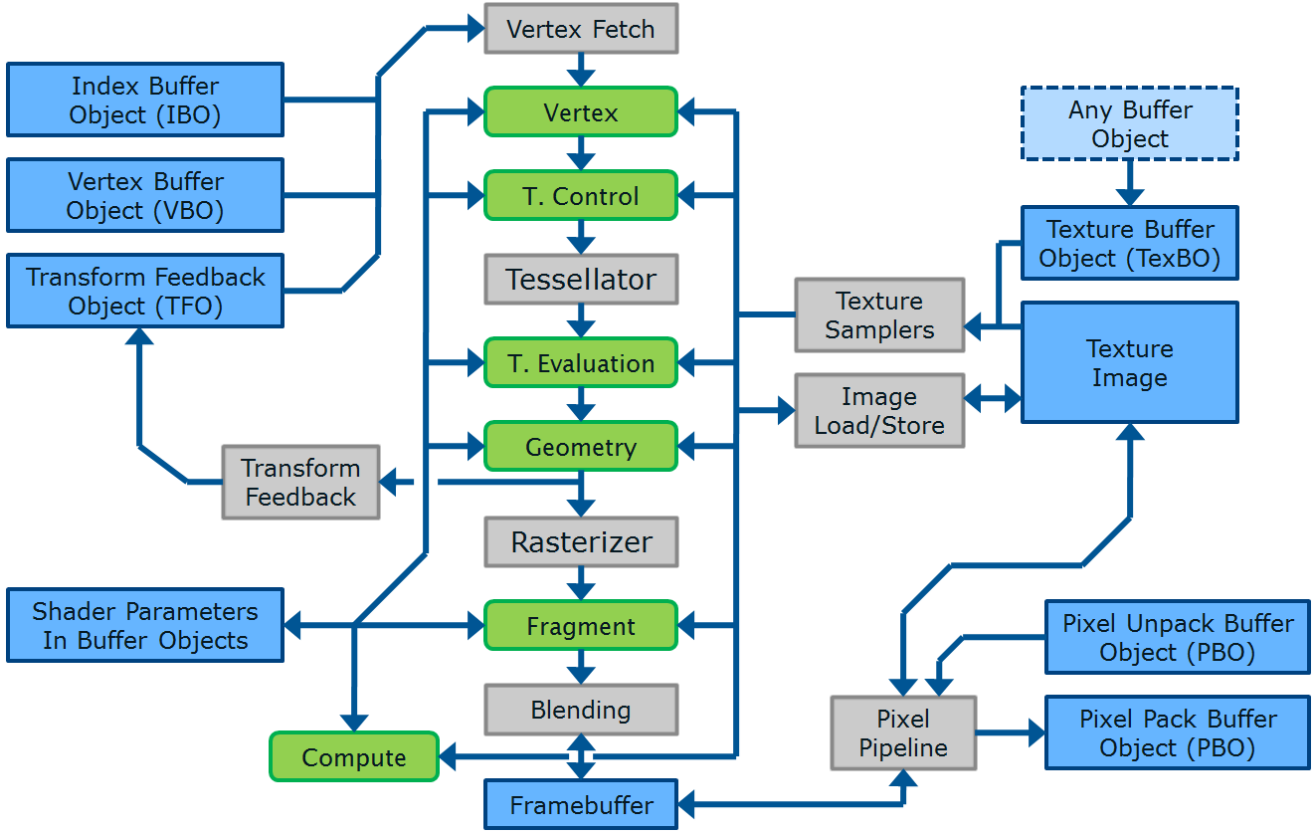


Tessellation

Tessellation

GPU Architecture - Rendering Pipeline

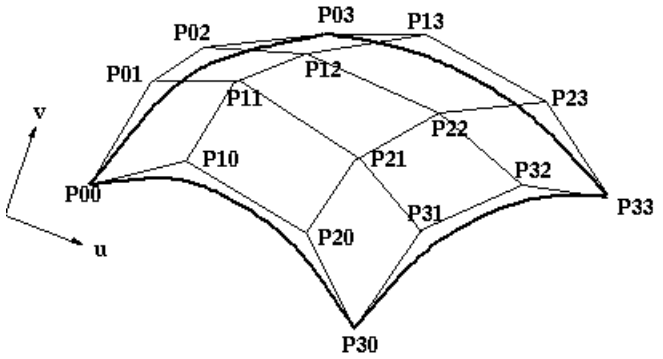
OpenGL 4.4



GPU Architecture - Rendering Pipeline

Patch

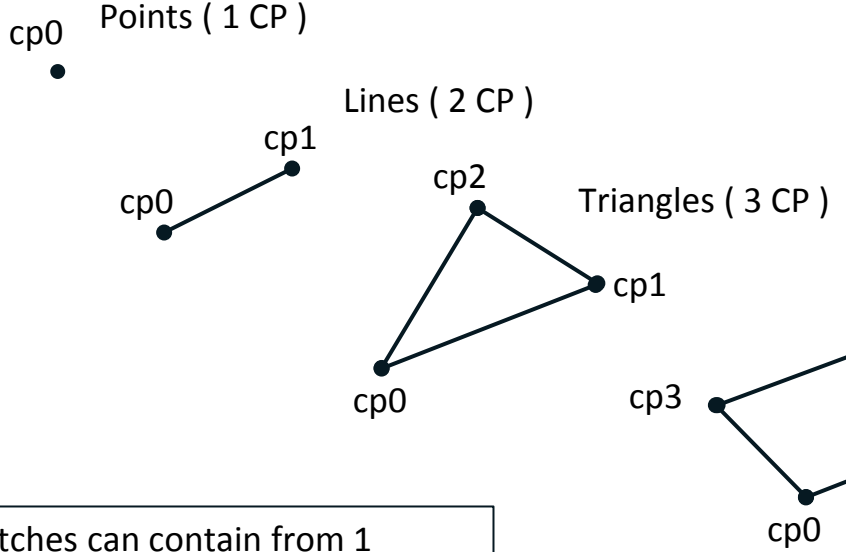
- PATCH is a generalized abstract description of surface or line
- PATCH allows to describe any simple primitive like point, line, triangle or quad
- PATCH can also describe complex surfaces like Bezier bi-cubic surface
- PATCH is constructed from Control Points
- Control Points helps describe shape of PATCH surface



- PATCHes are transparent and backward compatible
- We can treat basic input primitive as PATCH
- Then it's Vertices are treated like Control Points
- This allows to use tessellation without changes in mesh description
- In fact there is no difference between them, important is how you use them!

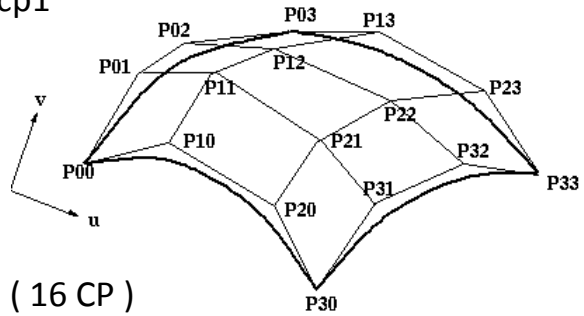
GPU Architecture - Rendering Pipeline

Patch types accepted by OpenGL:



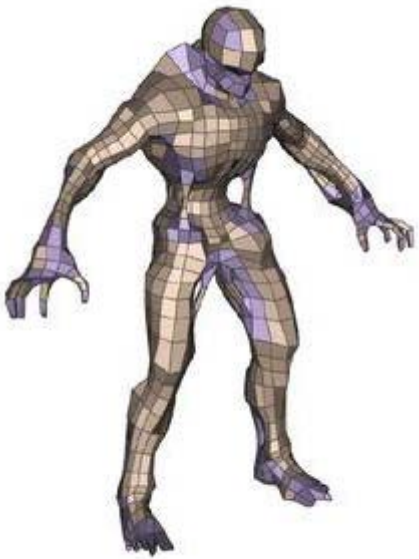
Separate Patches
(GL_PATCHES)

Patches can contain from 1
to up to 32 Control Points.



Bezier bi-cubic surfaces (16 CP)

GPU Architecture - Rendering Pipeline



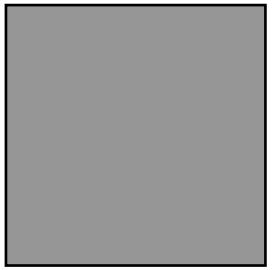
The "Imp" © Kenneth Scott, id Software 2008

Tessellation process

- Mesh is now build using Patches, in most cases Bezier bi-cubic surfaces with information about surrounding Patches, or simple Quad Patches.
- For each Patch Control Point, Tessellation Control Shader (or Hull Shader in DirectX) will calculate how much given Patch should be subdivided.
- Then Fixed Function Unit – Tessellator will generate different amount of triangles from each patch.

GPU Architecture - Rendering Pipeline

The "Imp" © Kenneth Scott, id Software 2008

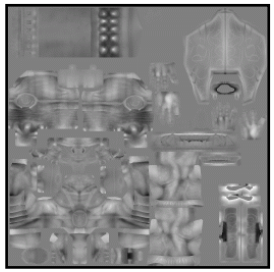


Tessellation process

- For each vertex of geometry produced by Tessellator, Tessellation Evaluation Shader (or Domain Shader in DirectX) will be executed.
- If there is no additional information available about details of geometry, TES can position new vertices in the way, triangles will be layed out in smooth gradients.
- Here displacement map is empty.

GPU Architecture - Rendering Pipeline

The "Imp" © Kenneth Scott, id Software 2008



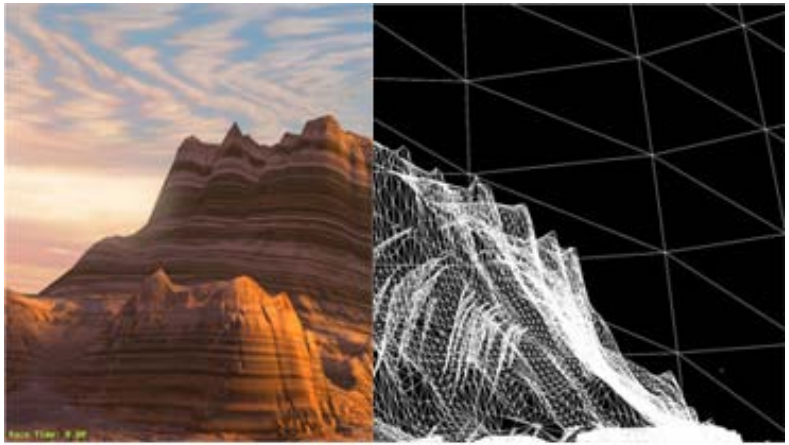
Tessellation process

- On the right we see „Imp" from Doom 3 created by Kenneth Scott.
- In this example we have available displacement map, which can be used by Evaluation Shader to displace vertexes in given directions, creating new detail.

GPU Architecture - Rendering Pipeline

Deformation, transformation

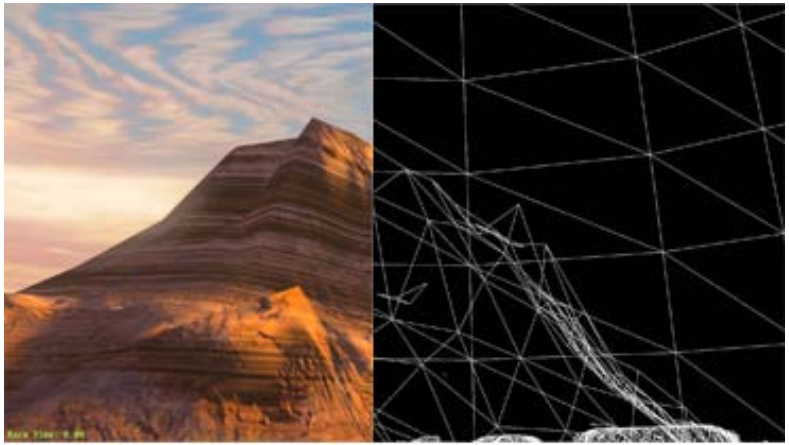
- Skin disease deformation
- Terrain modification and transformation
- Vehicles more realistic destruction



Tesselation On

Adaptive geometry

- Amount of geometry details dynamically matched to their visibility, distance from observer
- No need for few mesh versions

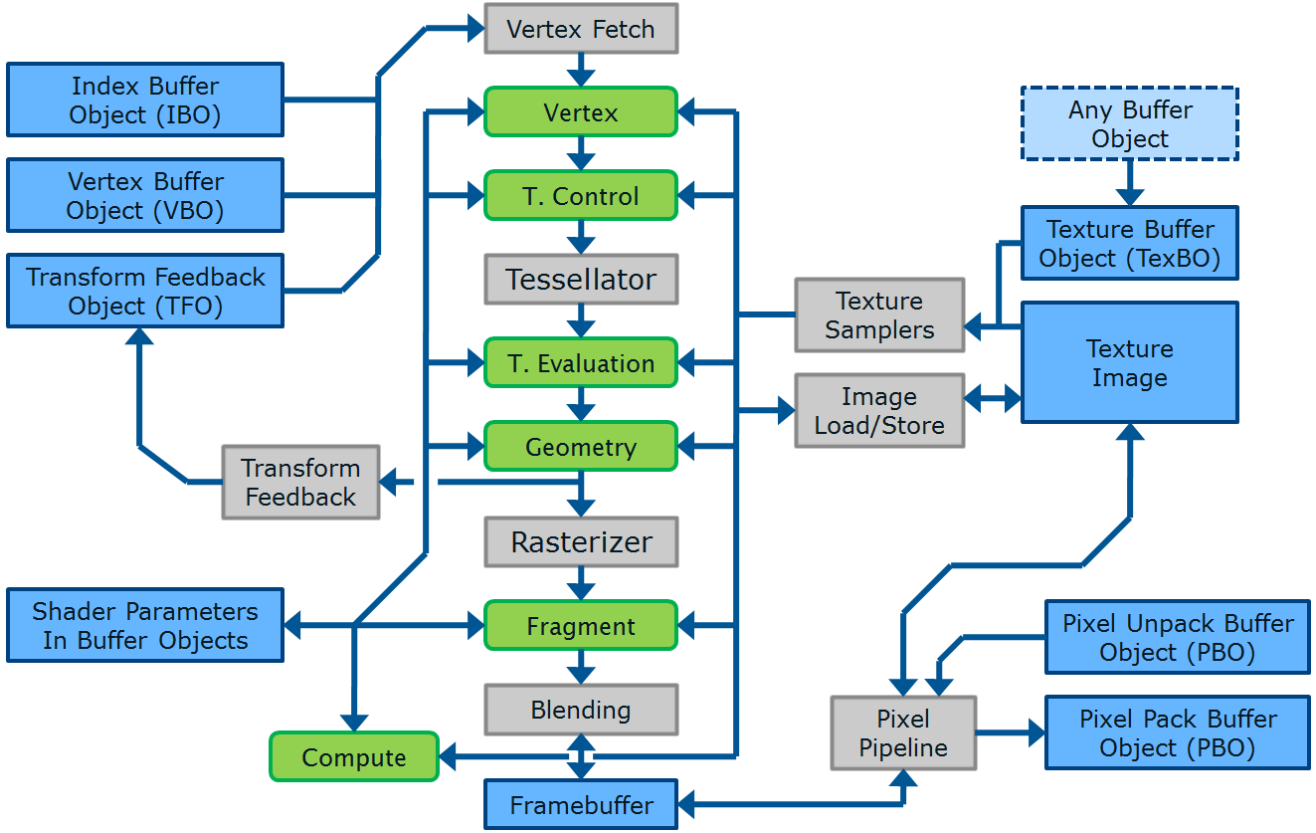


Tesselation Off

Geometry Shader

GPU Architecture - Rendering Pipeline

OpenGL 4.4

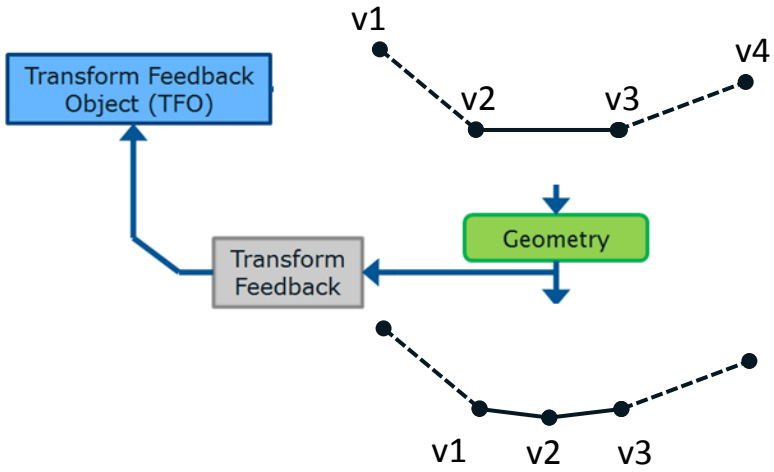


GPU Architecture - Rendering Pipeline

Making lines look smoother (perfect for hair):

IN: Separate Lines with Adjacency
(GL_LINES_ADJACENCY)

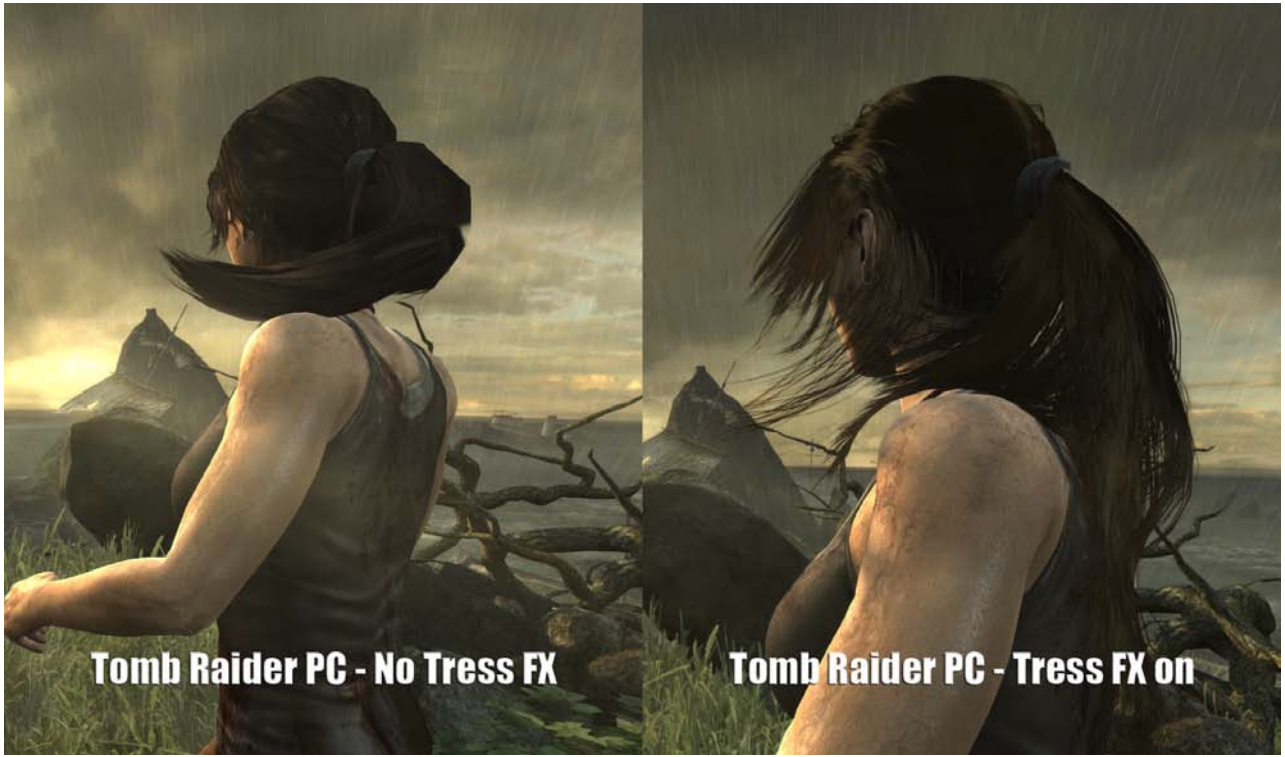
OUT: Line Stripes
(GL_LINE_STRIP)



Small set of lines is made more dense in Geometry Shader and modified to take into notice wind and motion. Resulting geometry is dense and represents hairs in current game frame.

GPU Architecture - Rendering Pipeline

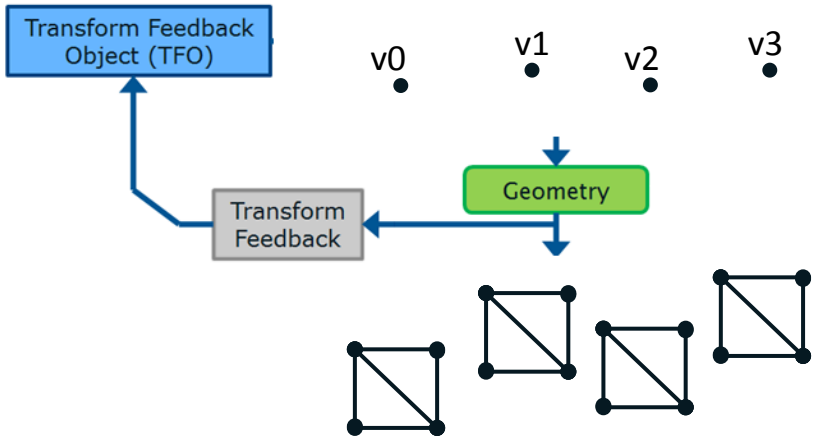
Making lines look smoother (perfect for hair):



GPU Architecture - Rendering Pipeline

Transforming set of points into quads (particles):

IN: Points
(GL_POINTS)



OUT: Triangle Stripes
(GL_TRIANGLE_STRIP)

Fire, water or other particles are stored as array of points. After processing their new positions, they are changed to quads on screen with mapped textures to them. Saving results through Transform Feedback allows continuous simulation.


UNREAL
ENGINE

GPU Architecture - Rendering Pipeline

Particle systems:

- Hair rendering
- Water, fire, smoke, dust, foam, clouds, explosion debris etc.

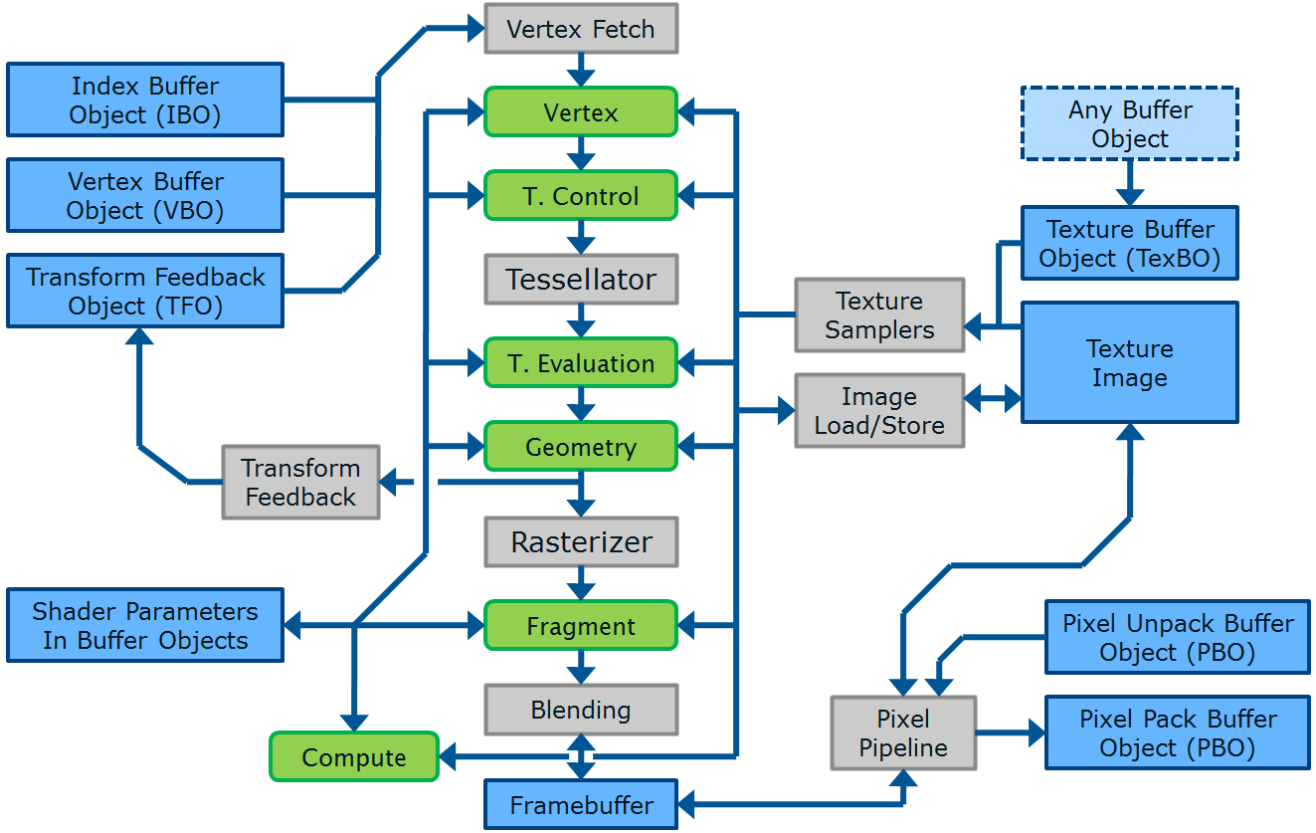
Geometry output:

- Allows procedural geometry by it’s gradual update in time
- General purpose algorithms on Hardware without Compute Shaders support


Fragment Shader

GPU Architecture - Rendering Pipeline

OpenGL 4.4

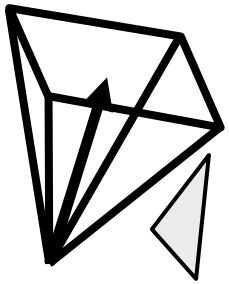


Rasterizer

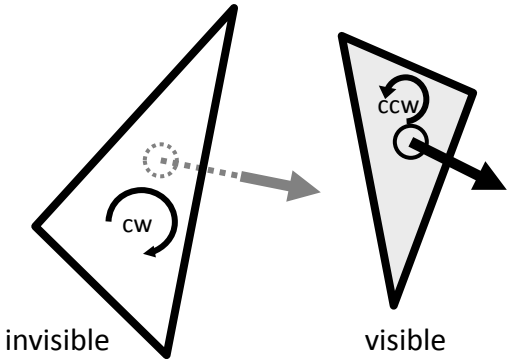
- Fixed-function unit.
- Bridge between geometry and pixel processing.
- Takes as an input graphic primitives transformed to screen space with their attributes.
- Performs culling, clipping and rasterization of graphic primitives into fragments. Rasterization is similar to image quantization.
- As a result spawns Fragment Shaders for all fragments of the Render Target that are covered by the graphic primitives.

Culling

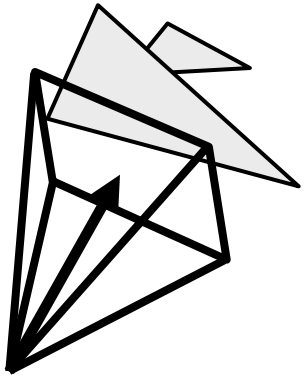
- Removes all primitives that:
- Are oriented away from observer (back face culling)
- Are farther than view cone reaches (early z culling)
- Are outside of view frustum (view culling)



View Culling



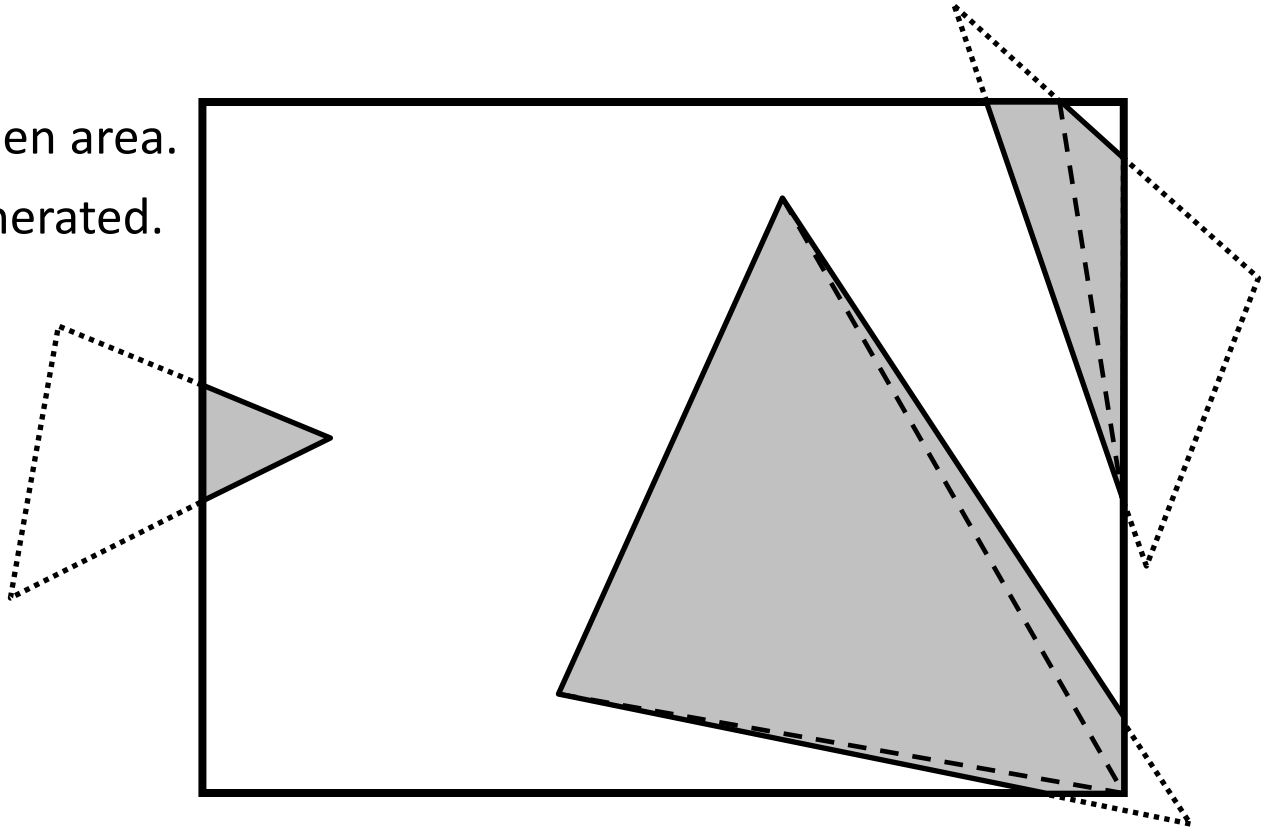
Backface Culling



Early Z Culling

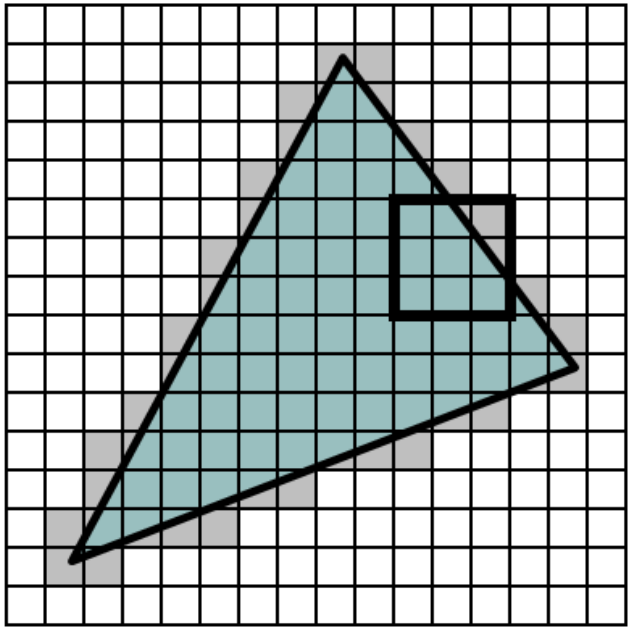
Clipping

- Cuts primitives to the screen area.
- New primitives can be generated.



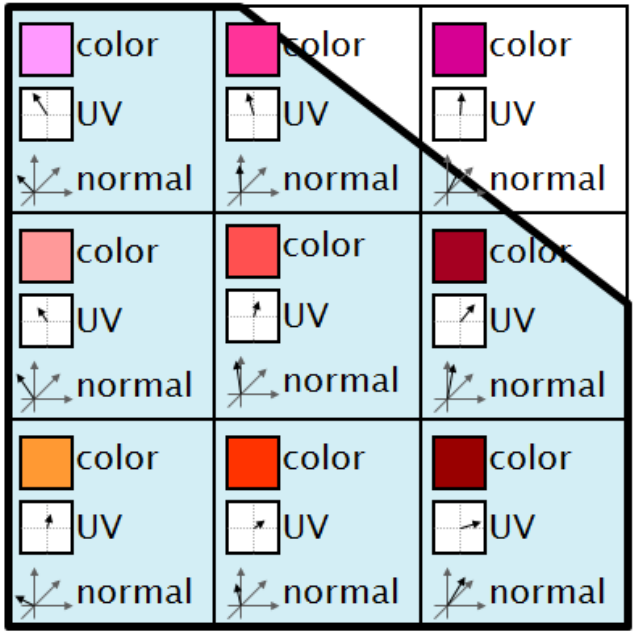
Rasterizing

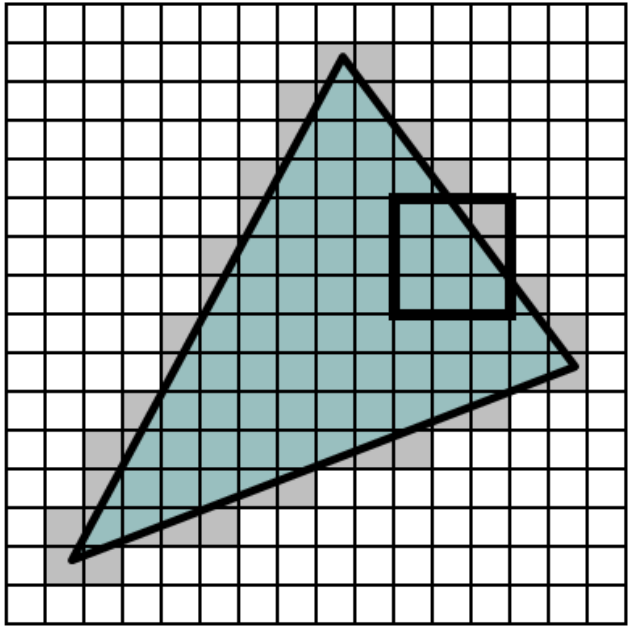
- Converting triangles into pixels.
- Vertex attributes from corners of triangle are interpolated for each pixel for which Fragment Shader will be spawned.
- These attributes can be color, texture coordinates or any other value.



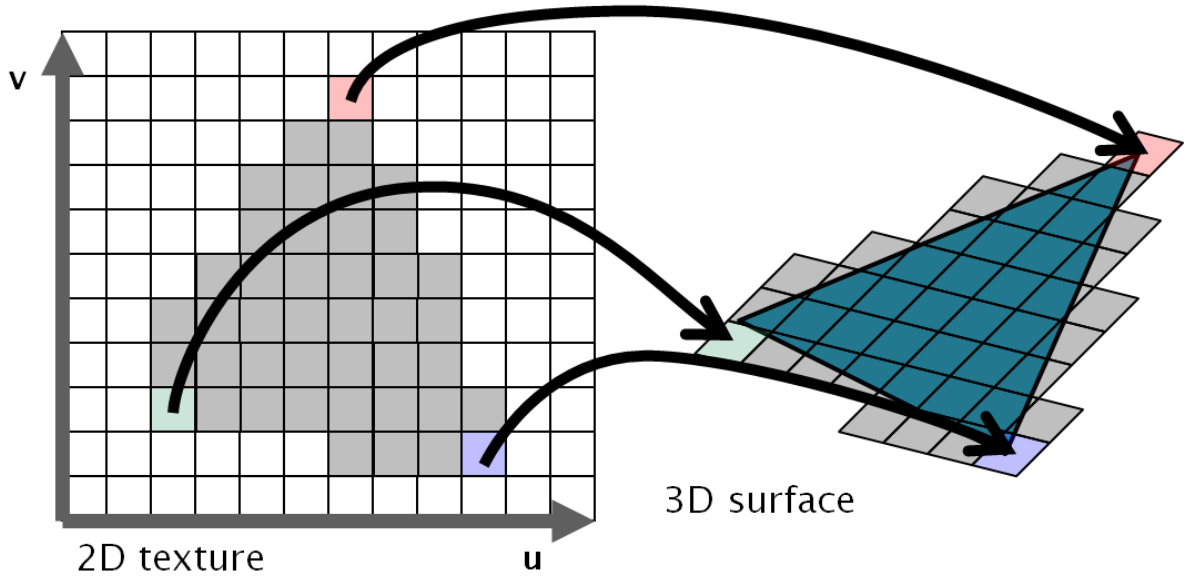
GPU Architecture - Rendering Pipeline

Geometry is now in screen space. It is time to convert it to pixels and compute their final colors.

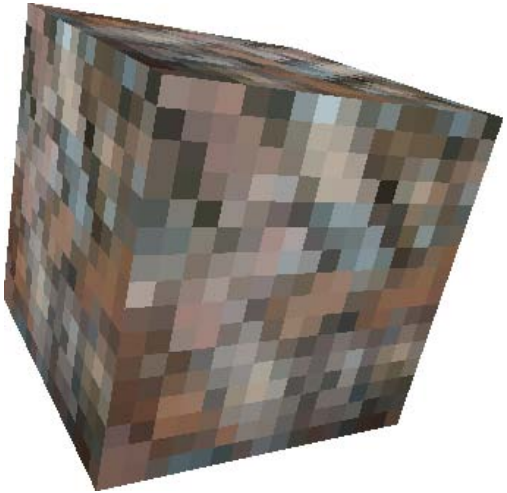




Texture Mapping



Texture Mapping

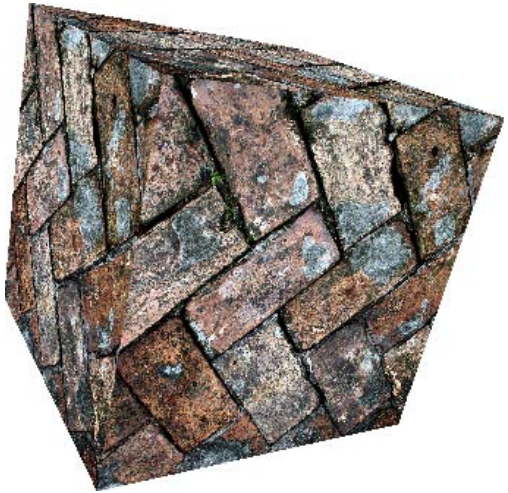


pixelization

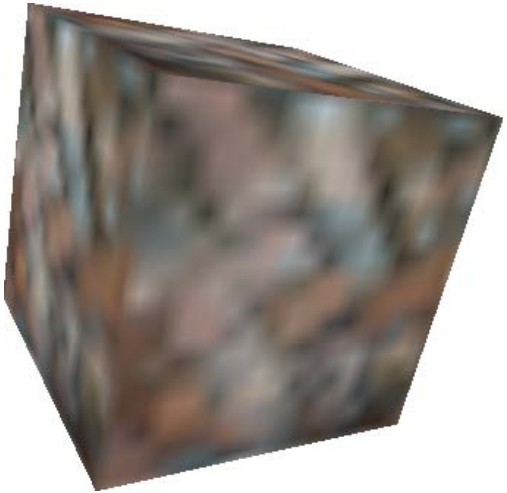
when low resolution texture is close to observer distinc texels become visible on texture mapped surface

aliasing

when high resolution texture is far from observer – moire is visible on texture mapped surface



Texture Mapping



pixelization's solution

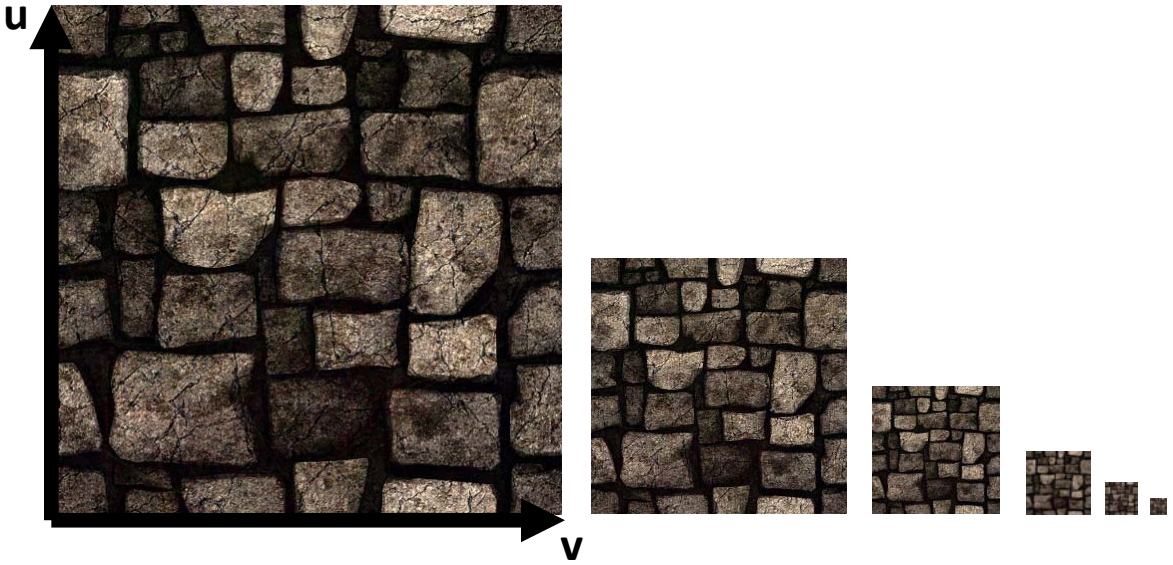
filtering: use weighted average when sampling texels

solution for aliasing

mip-mapping: storing texture at multiple resolutions and choosing correct one for each pixel (filtering helps too)

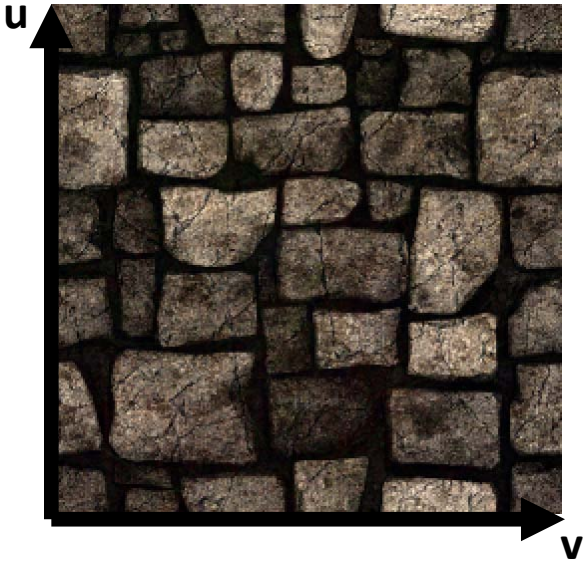
MipMaps

- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



MipMaps

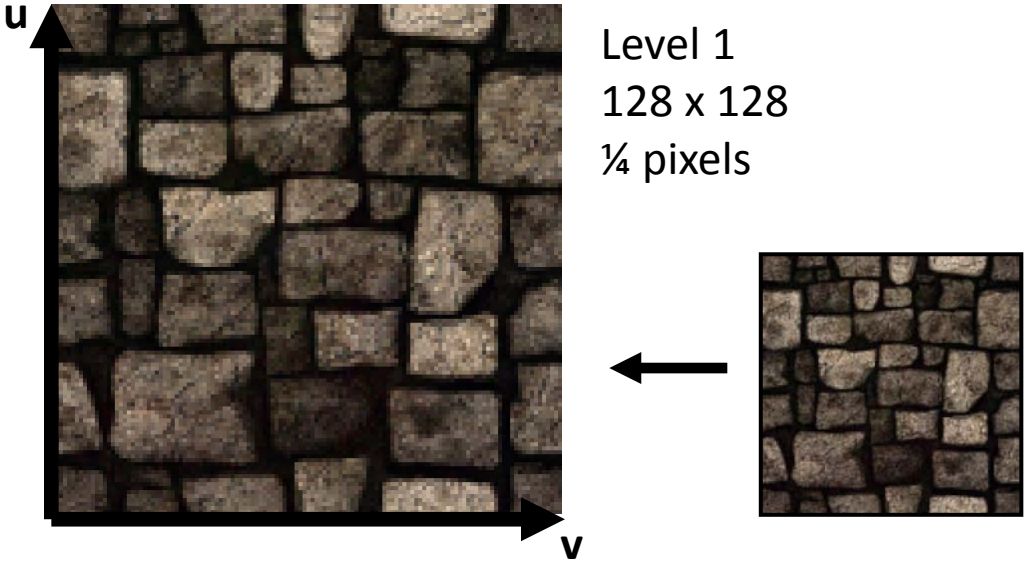
- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



Level 0
256 x 256
Original size

MipMaps

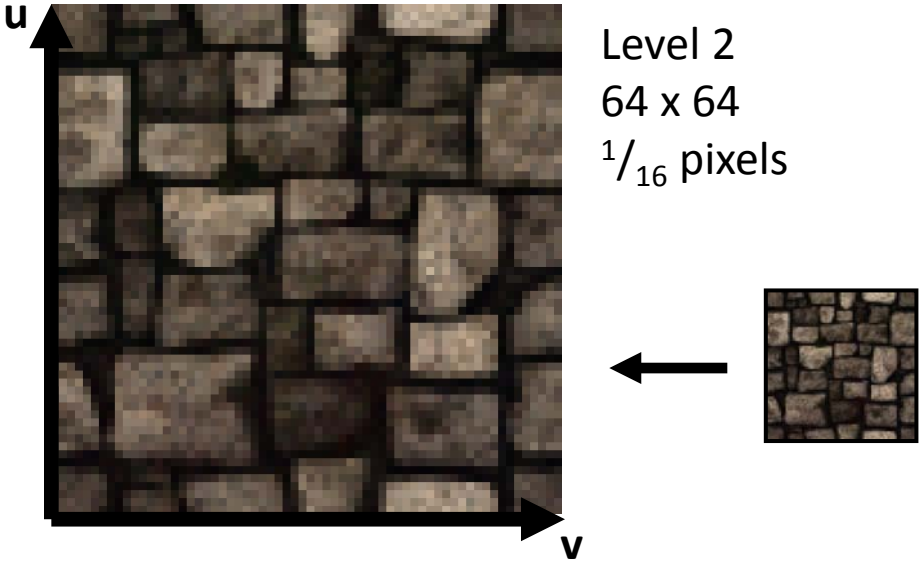
- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



Level 1
128 x 128
¼ pixels

MipMaps

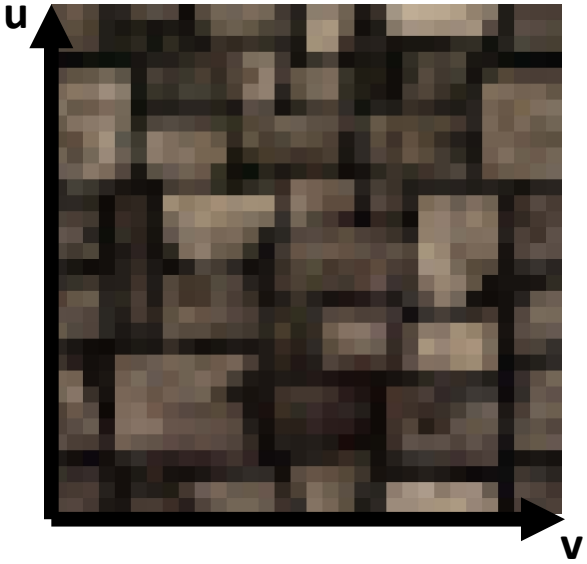
- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



Level 2
64 x 64
$^{1}/_{16}$ pixels

MipMaps

- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



Level 3
32 x 32
$^{1}/_{64}$ pixels

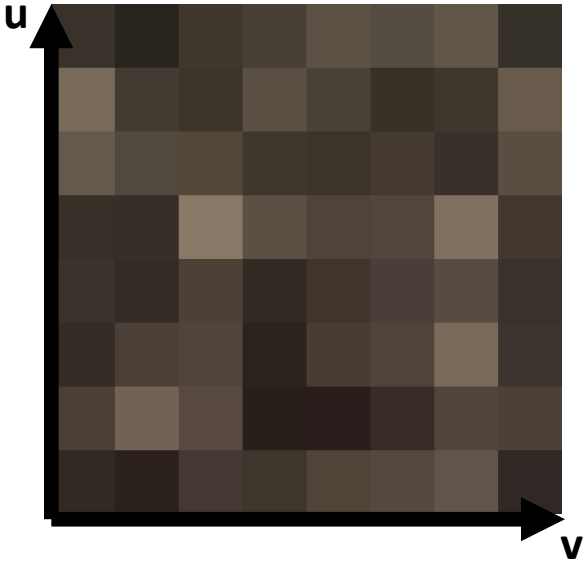
MipMaps

- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.

Level 4
16 x 16
$^{1}/_{256}$ pixels

MipMaps

- Require additional 33% of memory.
- Solves serious sampling problems related to memory locality reads.



Level 5
8 x 8
$^{1}/_{1024}$ pixels

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

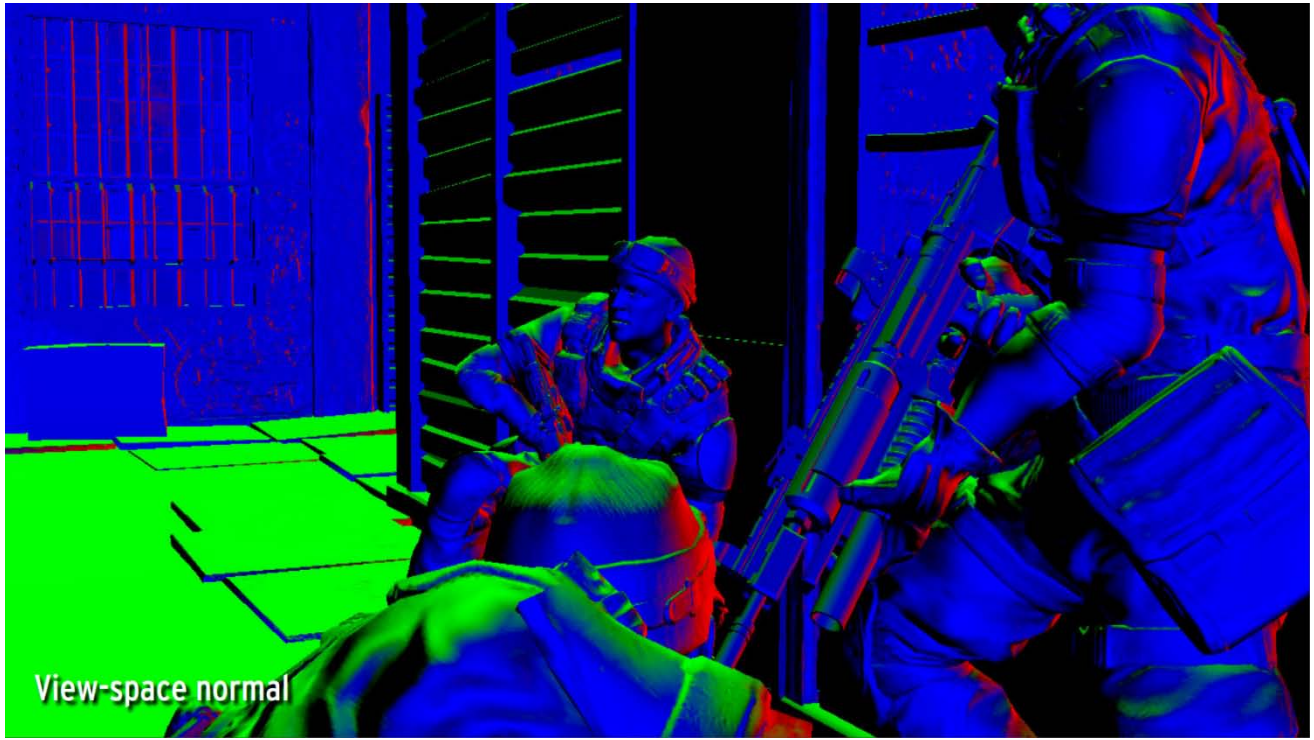


Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

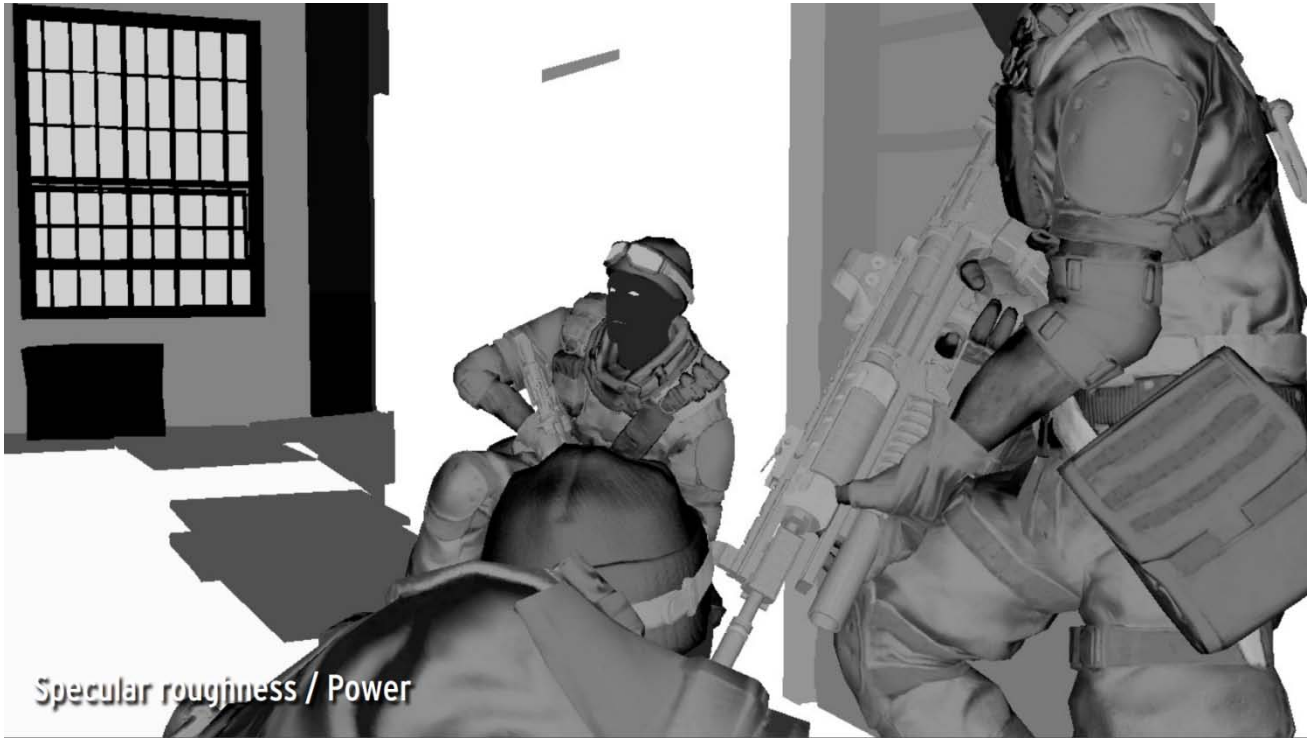


Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

Image source: „Deferred Rendering in Killzone 2” Guerilla Games

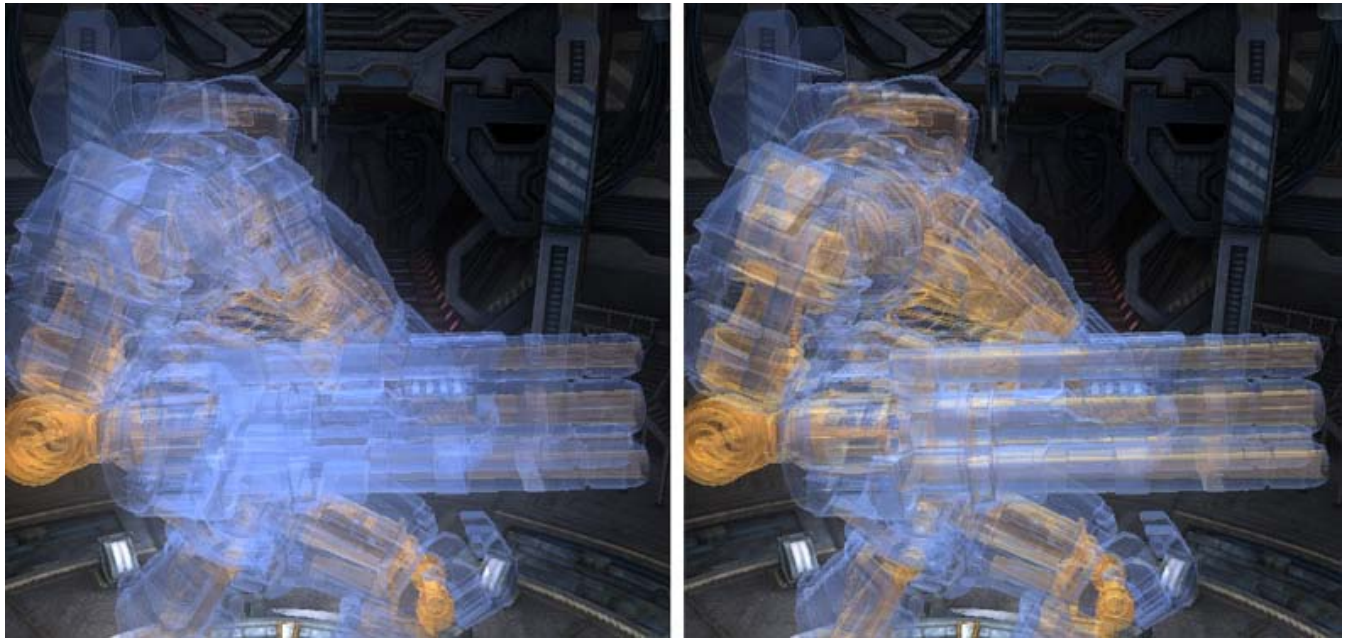
GPU Architecture - Rendering Pipeline

Geometry parameters are interpolated for each pixel.

Image source: „Deferred Rendering in Killzone 2” Guerilla Games

GPU Architecture - Rendering Pipeline

Since DirectX 11 and OpenGL 4.0, thanks to support of UAV's / Images, it is possible to calculate so called "Order Independent Transparency"



With OIT

Without OIT

GPU Architecture - Rendering Pipeline

Colors Calculation

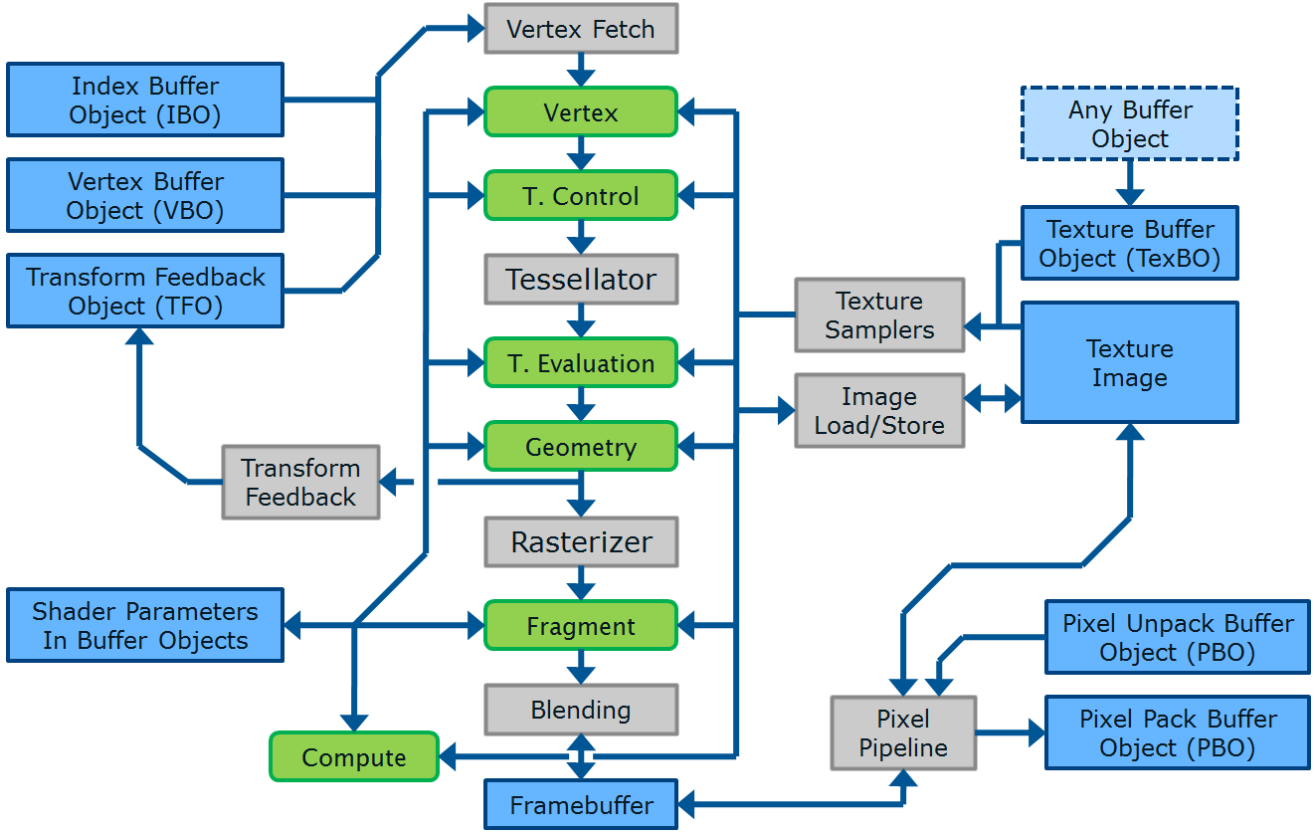
- Shading surfaces of geometry visible on screen, to enhance visual quality of rendered image.

- Simple texture mapping
- Increasing surface details
- Illumination models
- Procedural algorithms


Compute Shader

GPU Architecture - Rendering Pipeline

OpenGL 4.4



GPU Architecture - Rendering Pipeline

Compute Shader

- Latest, universal stage.
- Works on data sets, not geometry.
- Introduced in DirectX 11 and OpenGL 4.0.
- Commonly used for post-process of generated frame, but can be used to anything like:
  - physics, AI, animation, procedural generation, etc.
- Executed once per dispatch element in the grid.
- Dispatch grid can have up to 3 dimensions.
- It takes as an input textures, parameters and atomic counters and stores results in textures and buffers.

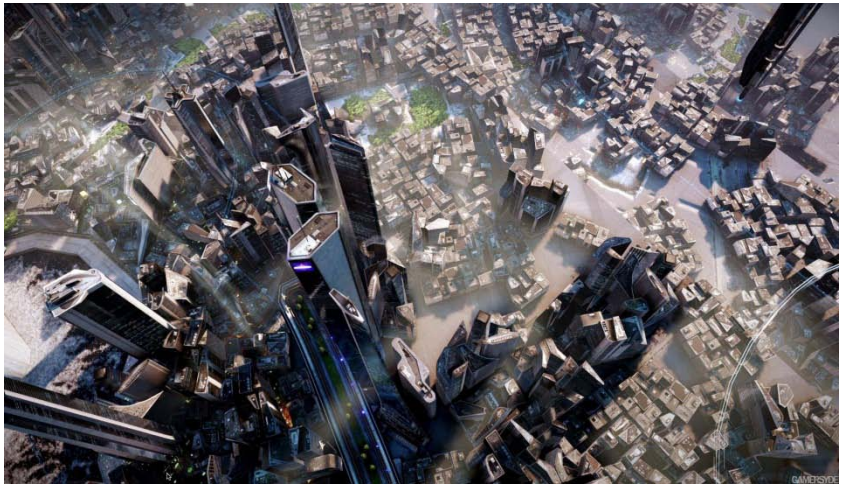

POST PROCESSING
Unprocessed image


POST PROCESSING
Cinematic color grading & depth of field

GPU Architecture - Rendering Pipeline

Post-processing effects

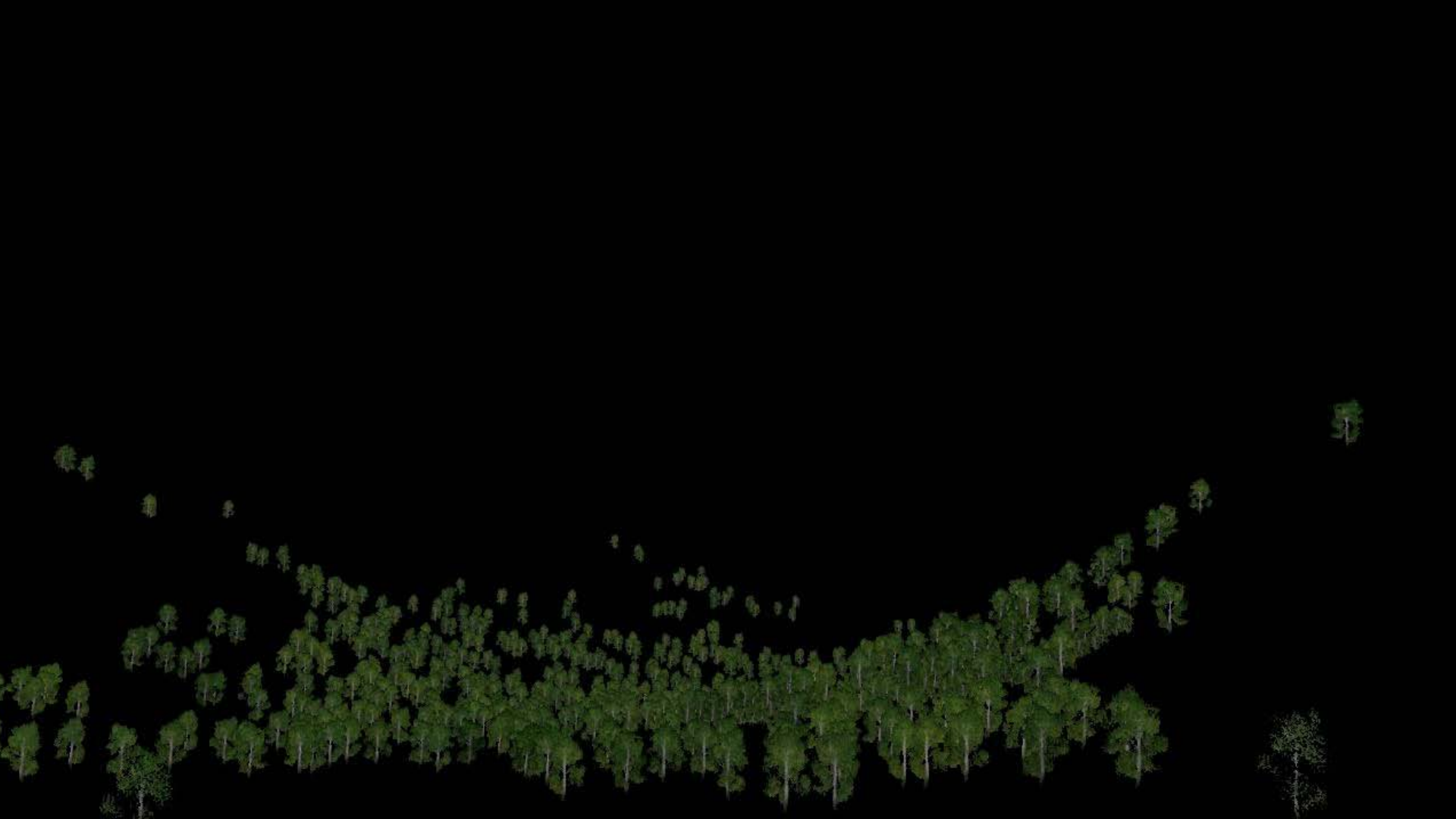
- Apply lighting techniques to enhance the mood in a scene. Secondary light bounces and Global Illumination.



Depth of field and defocus

- More realistic transitions of focal points – imagine looking through a gun sight or a camera lens. Bokeh DOF algorithms.

Conclusion



GPU Architecture - Rendering Pipeline

Pytania?