A **pixel** (picture element) represents a single "*point*" in a raster image. Its intensity has typically 3 components: **red**, **green**, and **blue**.

Graphics can be **raster**-type, where an image is an array of pixels, or **vector**-type consisting of encoding information about shape and color of an image.

A **primitive** represents a **basic unit** to create more complex objects. For example, sprites, text characters, or geometric shapes (**point**, **line**, and **triangle**) are primitives

The term **rendering** can be explained as **generating a 2D image from a 3D scene** by means of a computer. The scene can contain information about the geometry, camera, texture mapping, lighting model, shading effects, etc.

Rendering can be implemented by 3 methods

1. **scan-line** (also know as rasterization) algorithms
2. **ray-tracing** techniques
3. via solving the **rendering equation**

Methods 2 and 3 are realistic rendering that simulates 2 relevant aspects of light: [1] **transport** (how much light passes from one place to another), and [2] **scattering** (how surfaces interact with light). Method 1 uses **shading** (approximation of local light), such as the **Phong illumination model**, to decide the coloring of the rasterized image

One can add detailed color or patterns into a surface of a 3D model via **texture mapping**. Polygonal surfaces would require the addition of **texture coordinates** (also know as UV mapping) while parametric surfaces (such as non-uniform rational b-spline -NURB-) would have an intrinsic texture coordinate.

To add realism in shaders, there are various texture techniques (**mappings**) such as: normal, bump, parallax, specular, shadow, environmental, etc.

In raster images, an *artifact* called **aliasing** is presented by introducing low frequency information that is visually shown as noise in geometric edges or boundaries of textures. Some possible solutions are **supersampling**, **mipmapping**, or **texture filtering** (nearest-neighbor, bilinear, trilinear, anisotropic, etc.)

Curves
Can be defined as a **trajectory of a moving point**. It can be represented as follow

	Explicit $y = f(x)$	**Implicit** $f(x,y) = 0$	**Parametric** $q(t) = [x(t), y(t)]$
Circle	$\sqrt{r^2 - x^2}$	$x^2 + y^2 - r^2$	$r[\sin(t), \cos(t)]$
Parabola	x^2	$y - x^2$	$[t, t^2]$

The **parametrization of a line** given 2 points, $\mathcal{A}$ and $\mathcal{B}$, can be thought as an initial **point** and a **direction** as

$$\text{Line}(t) = \mathcal{A} + t(\mathcal{B} - \mathcal{A}),\ t \in \mathbb{R}$$
$$\text{Line}(t) = (1-t)\mathcal{A} + t\mathcal{B}$$
$$\text{Line}(t) = [(1-t)x_0 + ty_0, \ldots, (1-t)x_n + ty_n]$$

The **derivative** of a curve is the **tangent vector** at a given point.

Given the **weights** a_i and the **basis functions** b_i, for $i \in [0, n]$, a **polynomial curve** can be represented as

$$q(t) = a_0 b_0(t) + \cdots + a_n b_n(t) = \sum_{k=0}^{n} a_k b_k(t)$$

For example, a line would be $b_0 = (1-t)$, $b_1 = t$, and $b_{i \geq 2} = 0$.

Relevant curves are **degree 3** (also know as **cubics**) given that 3 is the lowest degree that can represented an **S-shape** (shape with an inflection point). Since the cubic curve

$$q(t) = [f_0(t), f_1(t), f_2(t)],\ f_u(t) = x_u + y_u t + z_u t^2 + w_u t^3$$

suffers from too many coefficients to define a shape, a solution is to use **control points**.

Schema \ Basis	b_0	b_1	b_2	b_3
Bezier	$(1-t)^3$	$3t(1-t)^2$	$3t^2(1-t)$	t^3
Hermite	$2t^3 - 3t^2 + 1$	$-2t^3 + 3t^2$	$t^3 - 2t^2 + t$	$t^3 - t^2$

Ray Tracing
A color in the computer needs the discretization mapping

$$f = [0, 1) \mapsto [0, \ldots, 255] = i$$

Due to non-linearity on screens one might do a **gamma correction**, $i = \texttt{int}(256 * f^{\gamma})$

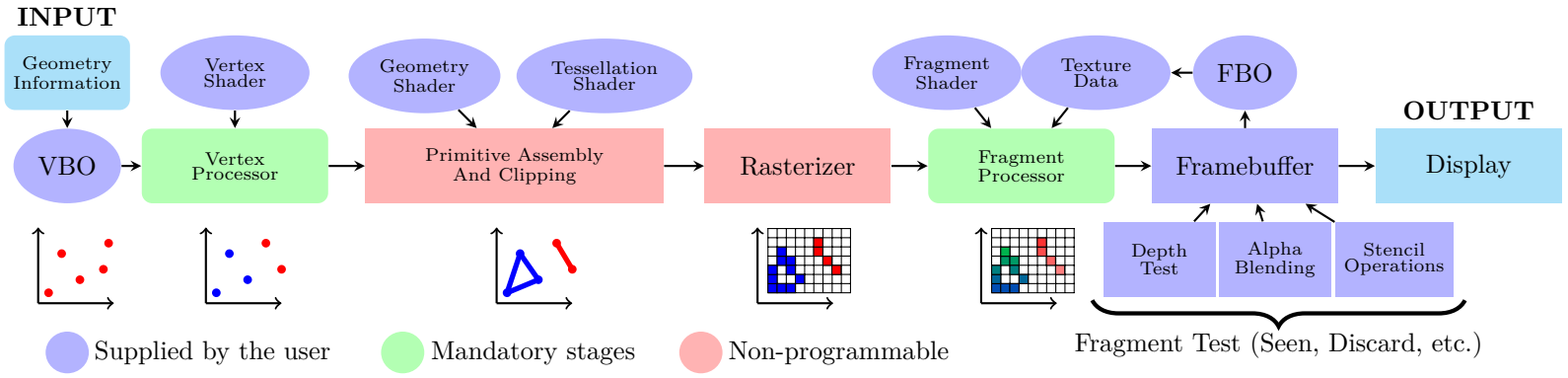
With vectors $\mathbf{u}$ and $\mathbf{v}$, relevant relations between them are

(Dot Product) $\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}||\,||\mathbf{v}|| \cos\theta$
(Cross Product Orthogonality) $(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u} = (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{v} = 0$
(Cross Product Norm) $||\mathbf{u} \times \mathbf{v}|| = ||\mathbf{u}||\,||\mathbf{v}|| \sin\theta$

Given the ray $\mathcal{R} = \mathbf{a}_r + t\mathbf{u},\ t \in [0, \infty)$ and an implicit surface $f(\mathbf{p}) = 0$ closed solutions for the **intersection surface-ray** are

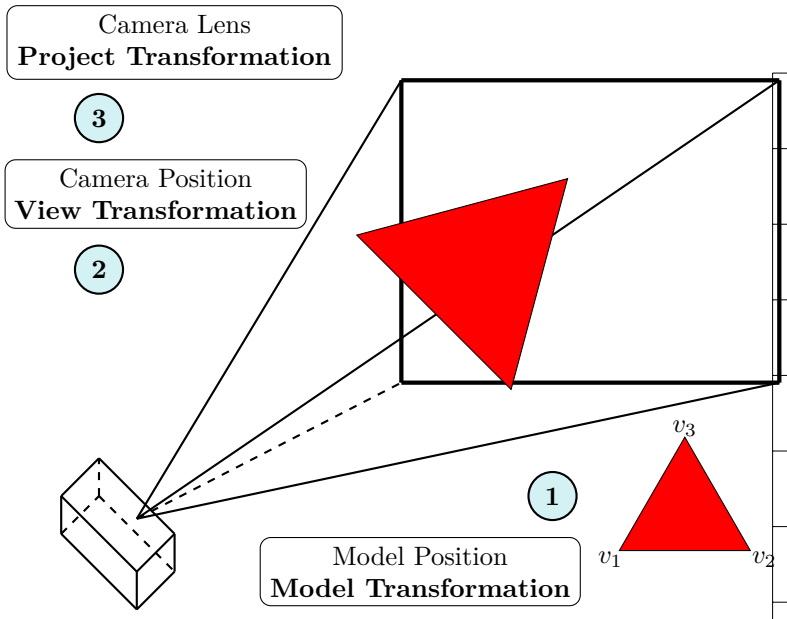
Surface	**Intersection**
Plane $(\mathbf{p} - \mathbf{a}_p) \cdot \mathbf{n} = 0$	$\frac{(\mathbf{a}_p - \mathbf{a}_r) \cdot \mathbf{n}}{\mathbf{u} \cdot \mathbf{n}}$
Sphere $\|\|\mathbf{p} - \mathbf{a}_c\|\|^2 - r^2 = 0$	$\frac{-\mathbf{u}\cdot\mathbf{b} \pm \left[(\mathbf{u}\cdot\mathbf{c})^2 - \|\|\mathbf{u}\|\|^2\left(\|\|\mathbf{c}\|\|^2 - r^2\right)\right]^{\frac{1}{2}}}{\|\|\mathbf{u}\|\|^2}$ $\mathbf{b} = (\mathbf{a}_r - \mathbf{u})$ $\mathbf{c} = (\mathbf{a}_r - \mathbf{a}_c)$
Triangle	

OpenGL Core Pipeline



1. **Vertex Processor**: Transforms vertex position/attributes (color, normal, texture)
2. **Primitive Assembly And Clipping**: Assembles data into primitives (points, lines, triangles)
3. **Rasterizer**: Determine which pixel locations are associated with each primitive
4. **Fragment Processor**: Decides appearance of each fragment (lighting/texture mapping)

Transformations



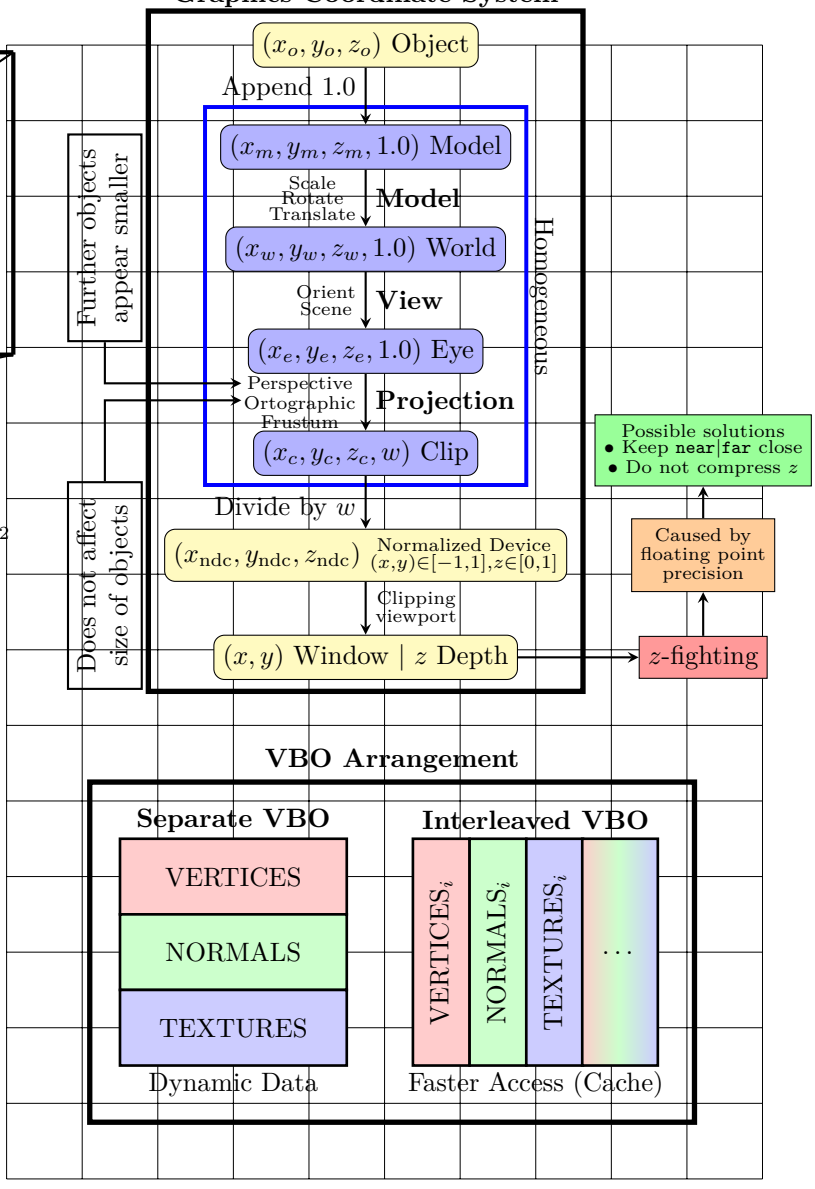
Using homogeneous coordinates is possible to express **translations**, **rotations**, **projections**, **scaling**, and **shearing** operations as linear transformations

$$\overbrace{\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Translation}} \overbrace{\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Scaling}}$$

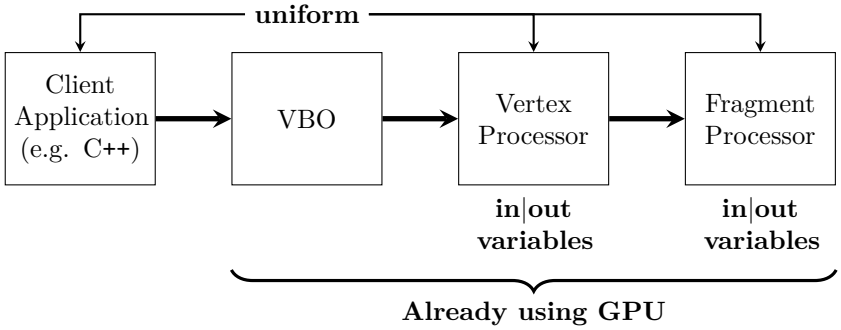
$$\overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\theta & -s\theta & 0 \\ 0 & s\theta & +c\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Rotation } X} \overbrace{\begin{bmatrix} c\theta & 0 & -s\theta & 0 \\ 0 & 1 & 0 & 0 \\ s\theta & 0 & +c\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Rotation } Y} \overbrace{\begin{bmatrix} c\theta & -s\theta & 0 & 0 \\ s\theta & +c\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{Rotation } Z}$$

$c\theta{=}\cos\theta,\ s\theta{=}\sin\theta$

Graphics Coordinate System

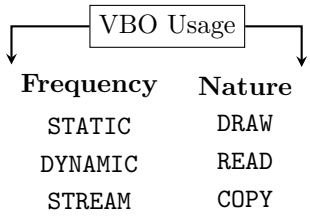


VBO Arrangement




```
uniform Matrices
{
  mat4 model;
  mat4 view;
  mat4 projection;
}
```

Usage example of `uniform`



e.g. `GL_STREAM_DRAW`

Client Side (CPU)

INITIALIZATION

```
GLint vao;
glGenVertexArray(1,&vao);
glBindVertexArray(vao);
GLfloat ** data = ... \\Initialize data;
glBindBuffer(GL_ARRAY_BUFFER,vbo);
glBufferData(GL_ARRAY_BUFFER,sizeof(data),data,GL_STATIC_DRAW);

    {GL_VERTEX_SHADER,"vs.glsl",GL_FRAGMENT_SHADER,"fs.glsl"}
GLuint programID = LoadShader(vertex,fragment);
glUseProgram(programID);

                                VERTEX SHADER POSITION   DATA TYPE        DATA STRIDE
glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,nullptr);
                                PER VERTEX # ELEMENTS    DATA NORMALIZATION   POINTER TO INIT MEMORY
```

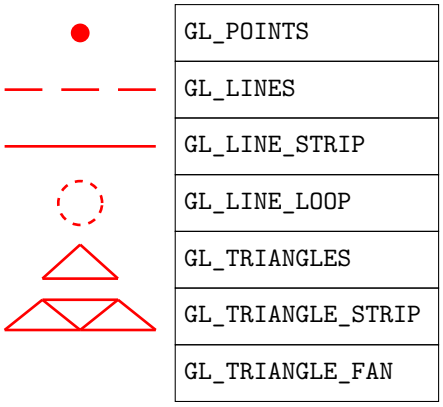
DISPLAY

```
glEnableVertexAttribArray(0);
glClear(GL_COLOR_BUFFER_BIT);
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES,0,number of vertices);
glSwapBuffers();
```

Server Side (GPU)

```
#version 330
layout (location = 0) in vec4 aPosition;
void main(){gl_Position = aPosition;}

#version 330
out vec4 fColor;
void main(){fColor = vec4(1,0,0,1);}
```



A **machine learning algorithm** is the process that shows underlying **relationship with the data**. Its outcome is a function F that outputs certain result provided an input. The function F is not a fixed function and can change depending the data injected.

For example, in the scenario of **image recognition** one can train an machine learning model that recognize a object in photos. The model in this case is a mapping between multiple dimensional pixel values and a binary value. The process of *discovering* this mapping (between pixels and yes/no answer) is what is called **machine learning** (ML).

Given that ML models are approximations, no model is 100% accurate, but state of the art ones (e.g. deep-learning approaches) can make fewer errors ($< 5\%$) than humans.

Three *learning* types,

- **Supervised**. Data sample contains a **ground truth** or target attribute y. The function F is one that takes non-target attributes X and outputs an approximation $\hat{y}$, i.e. $F(X) = \hat{y} \approx y$. Data with target attributes is commonly referred as **labeled**.
- **Unsupervised**. Some ways of learning without ground truth data are **Clustering** -samples into groups with similarities- (e.g. same color pixels, same shapes, same preference is listening, etc.), or **Association** -uncover hidden patterns among attributes- (e.g. someone listening podcast A also likes podcast B, someone buying item X also buys item Y, etc.)
- **Semi-supervised**. The data set is massive but the labeled samples are few (e.g. videos without category group/title, images with few of them segmented, etc.). To solve it one can combine previous approaches (supervised and unsupervised) to get results, for example if we can to predict images but only 10% of them are labeled one can do

  1. Apply supervised learning on the 10% to obtain a model and then use that model in the rest of the data
  2. Apply unsupervised learning via clustering to obtain groups of all the images and then apply supervised learning on each group individually.

Output of F can be divided in

Classification (for discrete values, such as boolean answers) . Given an image I with dimensions $W \times H$ and gray scale values in the range $[0, 255]$, the expected output of the classification model is a binary value True (1) or False (0)

$$F(I_{ij}) = 1|0,\ I[i][j] \in [0, 255], 0 \leq i < W, 0 \leq j < H$$

$$\text{Discrete} \xrightarrow[\text{modelA}: 1\%\ \ \text{modelB}: 50\%]{\text{Logistic Regression}} \text{Continuous}$$

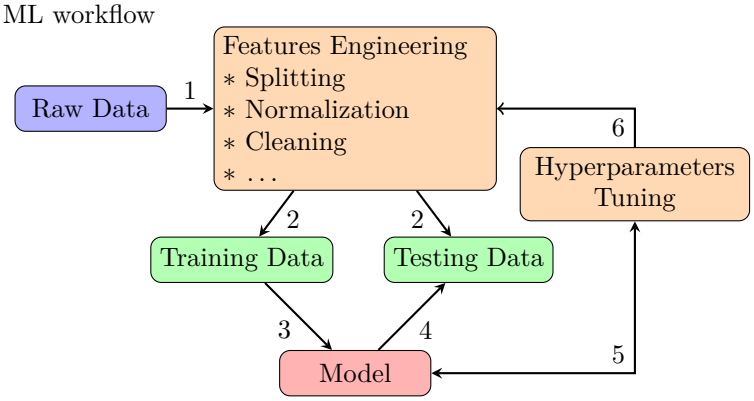
Regression (for continuous values, such as stock prices). Given data for real estate such as property surface, type of property (apartment, house, etc.), and location one expects to output a real value $p \in \mathcal{R}$. Each characteristic can be represented in a vector V that commonly is referred as **features**. Notice that one of those features is *categorical* (type of property) and would require a **transform** (unless on decision tree) to have a numerical representation.

$$F(V) = p \in \mathcal{R}$$

$$\text{Continuous} \xrightarrow[\text{buckets}]{\text{Price Ranges}} \text{Discrete}$$

Regression	Classification
Linear	K-means
Logistic	Gaussian Mixture
Decision Tree	Recommender
Support Vector Machine (SVM)	Non-Maximum Suppression

ML workflow



1. Decides what type of ML should be used: supervised or unsupervised? discrete or continuous?
2. Transforms data to be used in the ML algorithm. Some examples are **splitting** (commonly via 80/20 rule) in training and testing data sets; **augmenting** data such as rotations, scalings, shifting, etc; **encoding** categorical strings into numerical values; etc.
3. Creates the initial model (via training data + algorithm). It is called **training process**.
4. Test the model with the reserved testing data. It is called **testing process**.
5. Initial model commonly requires tuning to achieve higher confidence.
6. *Hyper* comes from the fact of manipulating external parameters that change the internal parameters of the model, for example in decision trees the **maximum height of the tree**; or how many items are processed, also known as **batch size**; etc.

In supervised algorithms there are two cases where the generated model does not fit well the data: **underfitting** (UF) and **overfitting** (OF).

Overall OF > UF since one can add **regularization** to OF steering it to a less complex model while fitting the data.

UF: Significantly deviated from ground truth. A cause is a over-simplified model for the data. **OF**: Little or no error thus does not generalize well to unseen data.A cause is over-complicated model for the data.

—**Missing Diagram**—

Bias: Tendency to *consistently* learn the same wrong thing.
Variance: Tendency to learn random things *unrelated* to the real signal.

Given a training set $S_t \coloneqq S_{\text{training}} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ a **learner** (ML algorithm) generates a model $\mathcal{F}$ and a **prediction** $\hat{y}_k = \mathcal{F}(\mathbf{x}_k)$ for a test sample $\mathbf{x}_k$, the **loss function** $\mathcal{L}(\mathcal{F}(\mathbf{x}_i), y_i)$ is the *cost* incurred by the difference between the prediction $\hat{y}_i$ and the true value y_i of the sample. One numerical example of a loss function is the **square error** $\mathcal{L} = ||\hat{y} - y||^2$.

For a set of training sets $S_n = \{S_t^1, \ldots, S_t^n\}$ and a loss function $\mathcal{L}$ the **main prediction** can be denoted as $y_m = min_{y'} \mathrm{E}_{S_n}(\mathcal{L}(y, y'))$, i.e. the prediction y' whose average loss with regards to all predictions $Y = \{\hat{y}_1, \ldots, \hat{y}_n\}$ is minimum. For the square error loss the main prediction reduces to **mean of the predictions** $y_m = \frac{1}{n}\sum_{i=1}^{n} \hat{y}_i$. Intuitively, the main prediction is the **general tendency** of the learner.

Bias can be mathematically defined as

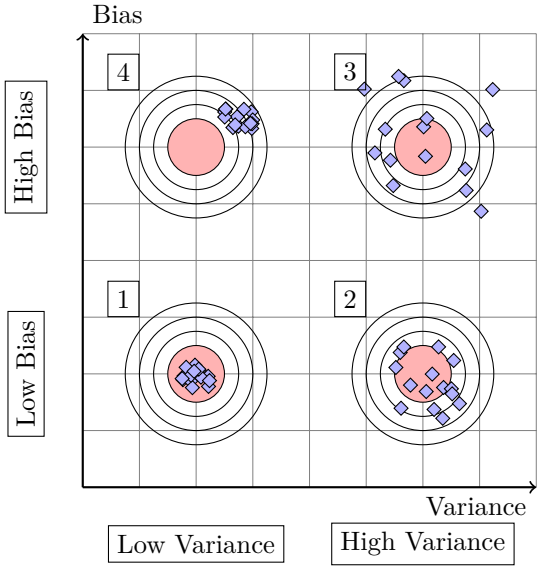
$$B(\mathbf{x}_i) = \mathcal{L}(y_m, t_i)$$

A high bias represents learning something fro the data that produces an erroneous prediction. On the other hand, a zero bias means the learner can produce models whose mean prediction is the desired target value.

Variance similarly can be mathematically defined as

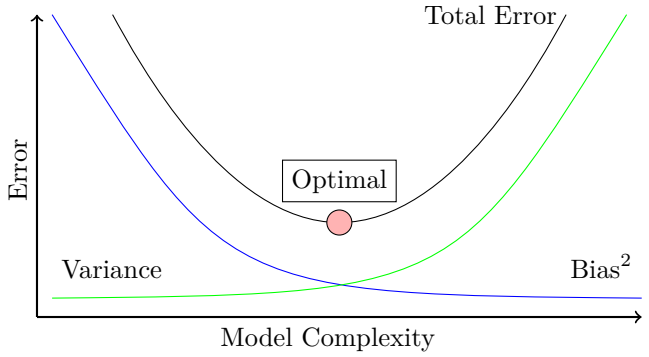
$$V(\mathbf{x}_i) = \mathrm{E}_{S_n}(\mathcal{L}(y_m), \hat{y})$$

The more stable the performance of a learner, the less its variance. Variance is **independent** of the true value of the predicted value, and zero is the learner always makes the same prediction regardless the training set S_n



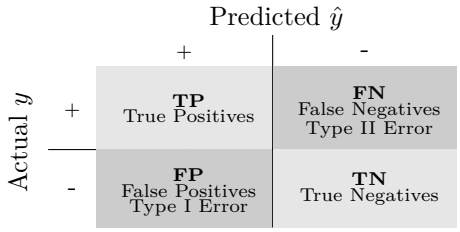
1. **Ideal learner**. Decent player that rarely miss the bullseye.
2. **Fair learner**. Scores some good points but too spread. Complex algorithms are in this bucket but can suffer from **overfitting**.
3. **Terrible learner**. Does not extract information from the data and learns nothing. Not different than making *random* guess.
4. **Naive learner**. Strategy too simple to capture essential information from the data. Algorithms in this bucket can suffer from **underfitting**.

There correlation between bias and variance is $\text{Error}(\mathbf{x}_i) = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$



Tuning parameters one can adjust bias and variance to find the optimal model.

In classification we can use **the confusion matrix** to assess the performance of a model



Metric	Formula	Interpretation
Accuracy	$\frac{\text{TP} + \text{TN}}{\text{T}_\text{N}^\text{P} + \text{F}_\text{N}^\text{P}}$	Overall performance of model
Precision	$\frac{\text{TP}}{\text{TP} + \text{FP}}$	Accuracy of **positive** predictions
Recall Sensitivity	$\frac{\text{TP}}{\text{TP} + \text{FN}}$	Coverage of actual **positive** sample
Specificity	$\frac{\text{TN}}{\text{TN} + \text{FN}}$	Coverage of actual **negative** sample

Some caveats about using machine learning as **silver bullet**

- Like humans, ML models make mistakes. Image recognition even being high accuracy (some around 95%) the model will continue to give false positives.
- It is hard, if not impossible, to correct mistakes made by ML in a case-by-case manner. An error in ML is not like a bug in software development.
- It is hard, if not impossible, to reason about certain ML models. For example, **ResNet-50** consists of 50 layers of neurons with 25.6 million of parameters.

Neural Network is an object that tries to mimic the human brain by using **various layers** that perform a particular task. Each layer consists of neurons that get activated under certain circumstances (simulating *stimuli*). Neuron are **inter-connected** between layers (via **weights**) which are powered by **activation functions**.

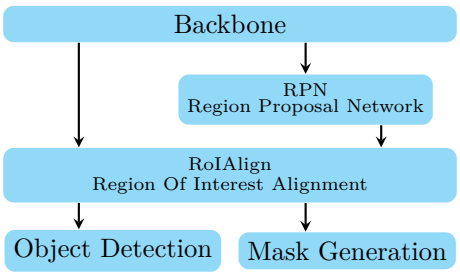
The information passing from one layer to another is called **forward propagation**. On the other hand, **back propagation** is the action of updating the weights to reduce the loss function via an optimizer (the most common one being **gradient descent**). A step to simulate input to output (across the data set) is denoted as **epoch**.

An activation function task is to transform a given input to a required output introducing non-linearities (without it the network would behave like a linear regression with limited learning and unable to be used in **images**, **videos**, **audio**, etc.)

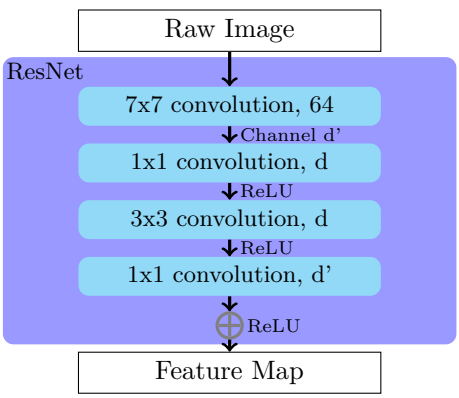
For Convolutional Neural Networks (**CNN**) some relevant layers

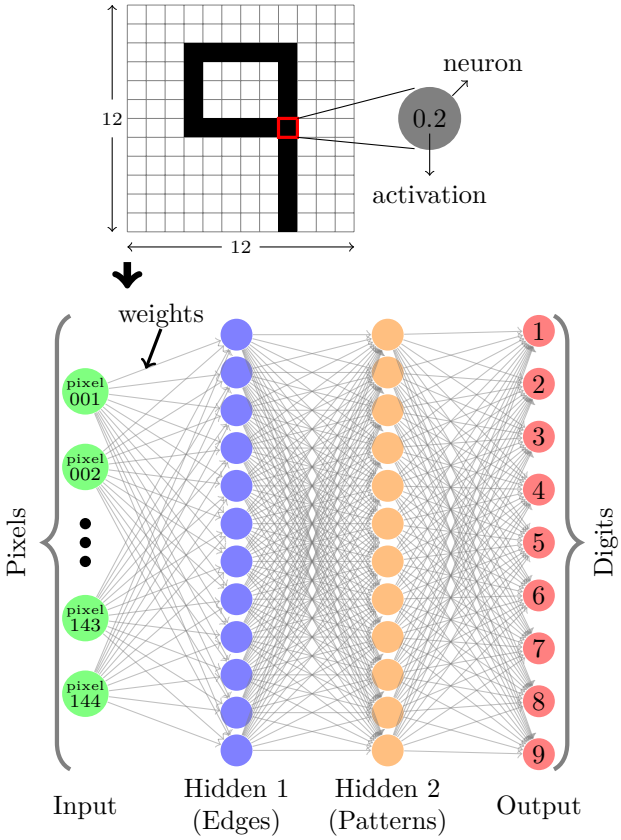
- **Pooling**. One *accumulates* features to reduce the spatial size of the network and reduce amount of parameters (e.g. max, average, general, etc.)
- **Convolution**. It uses a *kernel* to apply over an input signal and return relevant information from it (e.g. image processing, Fourier transform, moving averages, etc.)
- **Dropout**. Randomly replaced some of the elements of an input by 0 with a given *probability*.

Mask RCNN (Region-Based Convolutional Neuronal Network) is a state-of-the-art **deep learning** framework for instance segmentation in computer vision. It is done by adding *Fully Convolutional Networks* to *Faster R-CNN*

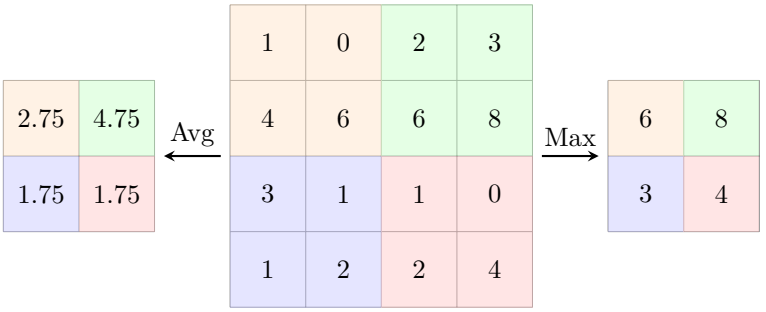


Backbone of Mask RCNN is the feature extractor (objects instances, classes, and spatial properties). Commonly it is composed of various **Residual Network** (ResNet).





A neural network visual representation for digit classification in image processing using two hidden layers.



Showing two possible pooling activation function.

Function	Definition	Advantages	Disadvantages
Identity	x	Easy to solve	Less power to learn
Binary Step	$\begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$	Simple binary classifier	Discontinuous?
Sigmoid	$\frac{1}{1+e^{-x}}$	Smooth Continuous Non-linear	Values in $[0,1]$ Vanishing gradient Non-symmetric and +
Tanh	$\frac{2}{1+e^{-2x}} - 1$	Same as Sigmoid	Symmetric over origin
ReLU	$\max(0, x)$	Simpler computation Sparsity Linearity	Exploding gradient Dying neurons
Leaky ReLU	$\begin{cases} ax & x < 0,\ a \in \mathrm{R} \\ x & x \geq 0 \end{cases}$	No dying neurons	Exploding gradient?
Softmax	$\frac{e_j^x}{\sum_{k=1}^{K} e_j^x} \quad j \in [1, K]$	Classifier > 2 classes Use of probabilities	Complex to compute?

Bit Manipulation

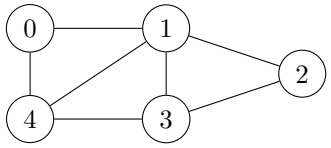
x	y	x & y	x | y	x ^ y	~x
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

Data Structures

- A **linked list** can be implemented as
  singly `1->2->3->NULL`
  doubly `NULL<-1<->2<->3->NULL`
  circular `1->2->3->1` or `1<->2<->3<->1`
- The **queue** can be done **circular**. The $n+1$ element is inserted in the 0 index if it is empty
- A **binary tree** (sometimes named only tree) is a hierarchical data structure that has **at most** 2 children
- Representation for binary tree is
  1. Data `node->data`
  2. Pointer to left child `node->left`
  3. Pointer to right child `node->right`
- Traversing a binary tree can be done as
  Depth First
  Inorder (Left-Root-Right)
  Preorder (Root-Left-Right)
  Postorder (Left-Right-Root)
  Breadth First
  Level order
- Maximum number of nodes of tree at level l is 2^l
- For a tree of height h, the maximum number of nodes is $2^{(h+1)} - 1$
- If not balanced the traversal complexity of a tree if height h can be $O(h)$
- A **binary search tree** (bst) has the following properties
  1. For each left subtree $\text{node}_l^{\text{value}} < \text{node}^{\text{value}}$
  2. For each right subtree has $\text{node}_r^{\text{value}} > \text{node}^{\text{value}}$
  3. The left and right subtree are also a bst
- A **graph** consist of
  1. Finite set of vertices called **nodes**
  2. Finite set of ordered pairs (u_i, v_j) called **edges**
  [Directed graphs have $(u_i, v_j) \neq (v_j, u_i)$]
  3. An edge may contain a **weight** (also named value or cost)
- For graphs it is common to represent
  $V \mapsto$ Number of vertices
  $E \mapsto$ Number of edges
- Graphs can be implemented as an **adjacency matrix** or as an **adjacency list**, having the following properties

Graph Implementation	Traversal	Space
Matrix	$O(V^2)$	$O(V^2)$
List	$O(V+E)$	$O(V+E)$

For example, given the following graph



Can be represented as the matrix (right) or the list (left)

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

```
0 [] -> 1 [] -> 4 []
1 [] -> 2 [] -> 3 [] -> 4 []
2 [] -> 1 [] -> 3 []
3 [] -> 1 [] -> 2 [] -> 4 []
4 [] -> 3 [] -> 1 [] -> 0 []
```

C Memory Layout

- Consists "typically" of 5 sections: 1. Text segment, 2. Initialized data segment, 3. Uninitialized data segment, 4. Heap, and 5. Stack
- The **text segment** contains the executable instructions. Can be places below the heap or stack regions. It is **read-only**
- **Initialized data segment** (or simply data segment) contains **global** and **static** variables **initialized by the programmer**. It can be subdivided in **read-only** and **read-write** areas. The following are examples of data segment
  `static int n = 10;`
  (defined anywhere)
  `const char * string = "hello world";`
  (defined outside the `main` function)
- **Uninitialized data segment** (or "block started by symbol" $\mapsto$ bss) contains **global** and **static** variables **not initialized explicitly** or initialized to zero. The following are examples of bss segment
  `static int i;`
  (defined anywhere)
  `int j;`
  (defined before the `main` function)
- The **stack** area contains the program stack (LIFO data structure) located in the higher parts of the memory (on x86 architecture it grows towards address zero). A set of values pushed for one function call is defined as **stack frame** (consisting at minimum of a return address). Each call to the stack allocates room for **automatic** and **temporary** variables. Allocation happens at **function call**
- The **heap** segment is where dynamic memory allocation takes place. Begins at the end of the bss segment and grows to larger addresses. In `C` language the heap area can be managed by `malloc, realloc` and `free` functions. "Contiguous" heap region is not mandatory for the previous C functions resulting in the possibility of **memory fragmentation**. Allocation happens at **execution of instruction**

C Memory Layout Comparison

	Stack	**Heap**
Memory	Allocated in contiguous block	Allocated in any random order
Allocation/Deallocation	Automatic by compiler	Manual by programmer
Cost	Less	More
Implementation	Easy	Hard
Access Time	Faster	Slower
Main Issue	Small memory	Memory fragmentation
Safety	Thread safe	Not thread safe (data visible to all threads)
Data Type	Linear	Hierarchical

Data Structures Complexity

	Accessing	Search	Insertion	Deletion	Space
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack (LIFO)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue (FIFO)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Hash Table		$O(1)$	$O(1)$	$O(1)$	$O(n)$
Binary Search Tree*	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(n)$
Binary Heap	Min Heap $O(1)$ Max Heap $O(1)$	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$	$O(n)$

*h would be the height of the tree. If tree is balanced $h = log\ n$

Dynamic Programming

Programming paradigm to **efficiently** explore all possible solutions to a problem. The problem should have one of the following characteristics

- Problem can be broken down in **overlapping subproblems**

  A rule of thumb is to notice if **future decisions depend on earlier decision**
- Problem has an **optimal substructure**

  For example the problem would ask optimal value (min or max) of something, here are some sample phrases
  - What is the minimum cost of...
  - Compute the maximum profit from...
  - How many ways can you...
  - Find the longest possible...

Care has to be taken to not confuse a dynamic programming (DP) problem with **greedy** problems. Both would have optimal substructure but greedy does not have overlapping characteristic.

Two ways of implementing a DP algorithm,
Top-Down (recursion) or **Bottom-Up** (iterative).

An important optimization in the top-down approach is **memoization** that stored previously computed subresults (commonly in a hash) to be re-used in the future, avoiding recomputation.

	Top-Down	**Bottom-Up**
Pro	Easy to write	Runs faster
Con	Overhead	Order of operations matter

In DP one can define a **state** as a **set of variables that can sufficiently describe a scenario**.

To "*produce*" an algorithm for a DP problem, one can follow the next steps:

1. Create a **function** or **data structure** that will compute the answer of the problem for **every given state**
2. A **recurrence relation** to transition between states
3. Define **bases cases** to avoid inifinite recursion or faulty iterations

For example, the following **climbing stairs** problem

> You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. **How many distinct ways** can you climb to the top?

Thus, the DP steps for above problem would be

1. **Function** $dp(i)$, where i represents how many ways to climb to the i^{th} step.
2. **Recurrence** $dp(i) = dp(i-1) + dp(i-2)$, since according to the description one can climb either 1 or 2 steps each time.
3. **Base Case** $dp(1) = 1$ and $dp(2) = 2$, since there is 1 way (1 step) to climb to step 1 and 2 ways (1 step + 1 step and 2 step) to climb to step 2.

A minimal implementation in C++ of above steps is

```
int dpRecursive(int i)
{
    if (i <= 2) return i; // base case
    // recurrence
    return dpRecursive(i - 1) + dpRecursive(i - 2);
}
int countSteps(int n) { return dpRecursive(n); }
```

Although the code would solve the problem, it has poor complexities, $\mathcal{O}(2^n)$ time and $\mathcal{O}(n)$ space.

To improve time complexity, **memoization** is required

```
unordered_map<int,int> memo;
int dpRecursiveWithMemo(int i)
{
    if (i <= 2) return i; // base case
    if (memo.find(i) == memo.end())
    {
        // recurrence
        memo[i] = dpRecursive(i - 1) + dpRecursive(i - 2);
    }
    return memo[i];
}
int countSteps(int n) { return dpRecursiveWithMemo(n); }
```

Which converts the time complexity to $\mathcal{O}(n)$.

One can also change the above **Top-Down** to a **Bottom-Up** implementation. The key for it is just to modify the recursion for an iteration while preserving the 3 steps to solve a DP problem, i.e.

1. **Data Structure** `dp[i]`, where `dp` is an array
2. **Recurrence** `dp[i] = dp[i - 1] + dp[i - 2]`
3. **Base Case** `dp[1] = 1`, `dp[2] = 2`

Thus, an implementation of the bottom-up version would be

```
int countSteps(int n)
{
    if (n <= 2) return n;
    vector<int> dp(n + 1, 0); // data structure
    dp[1] = 1; // base case
    dp[2] = 2; // base case
    for (int i = 3; i <= n; i++)
    {
        dp[i] = dp[i - 1] + dp[i - 2]; // recurrence
    }
    return dp[n];
}
```

Notice how for this problem `dp` requires to be of length $n + 1$. This implementation would have the same complexities as before, i.e. $\mathcal{O}(n)$ for time and space.

Lastly, one can improve the space complexity to be constant by noticing that at each step the iterative approach only cares about the **previous 2 steps**. Therefore, removing the array `dp` the **optimized solution for the climbing stairs problem is**

```
int countStepsOptimized(int n)
{
    if (n <= 2) return n;
    step1 = 1; // base case
    step2 = 2; // base case
    for (int i = 3; i <= n; i++)
    {
        int nextStep = (step1 + step2); // recurrence
        step1 = step2;
        step2 = nextStep;
    }
    return step2;
}
```

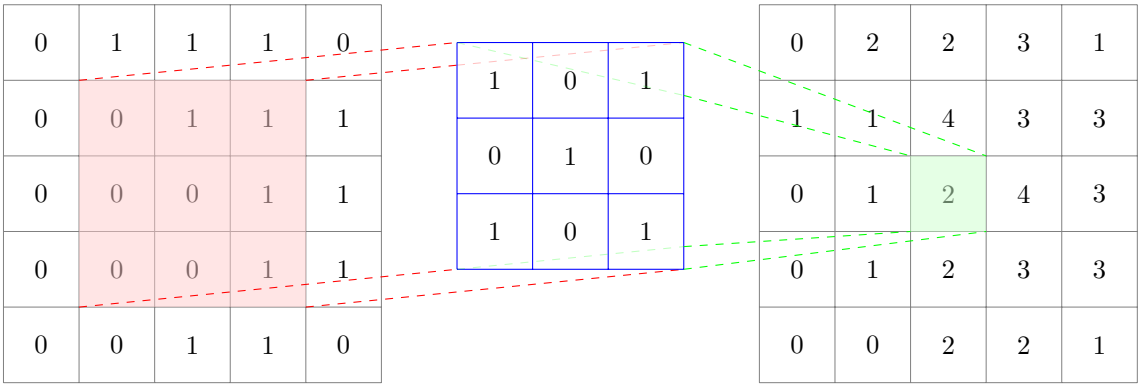
Which makes a $\mathcal{O}(1)$ space complexity.

Convolution is a mathematical operation on two function ($f$and g) that produces a third function ($f * g$) that expresses *how the shape of one is modified by the other*. Formally,

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)\, g(t - \tau)\, d\tau$$

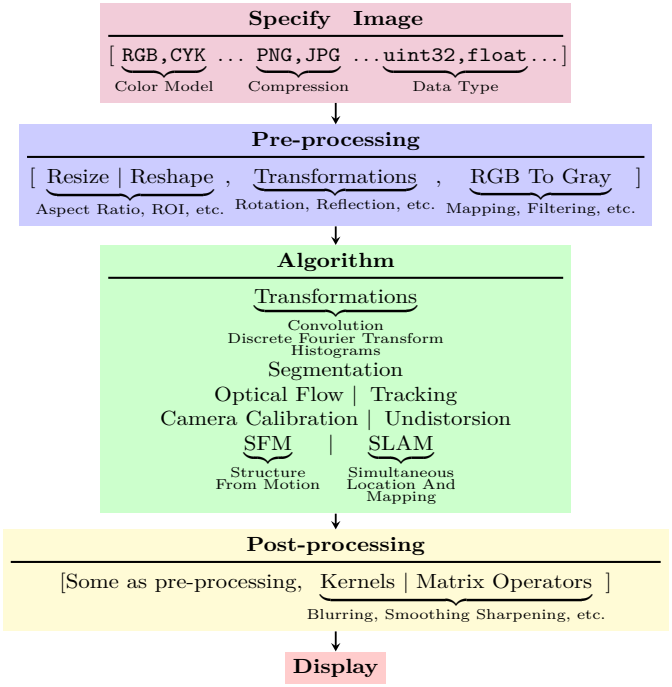
It can be generalized to n dimensions and in practice the integral can be changed for a summation ($\sum$) and the integral limits defined by the problem's context to do a **discrete convolution**. For example, a 2D convolution in image processing would be

$$\text{Conv2D}(x, y) = \sum_{i=-\frac{n}{2}}^{\frac{n}{2}} \sum_{j=-\frac{m}{2}}^{\frac{m}{2}} h(i, j)\, I(x - i, y - j)$$




```
bool outOfBounds(int n, int m, int w, int h)
{
  return (n < 0 || n > h - 1 || m < 0 || m > w - 1);
}
typedef vector<vector<int>> vvint;
vvint convolution2D(const vvint & M, const vvint & K)
{
  vvint convolution;
  int height = M.size();
  int width = M[0].size();
  int n2 = K.size() / 2;
  int m2 = K[0].size() / 2;
  convolution.resize(M.size());
  for (int n = 0; n < height; n++)
  {
    convolution[n].resize(width);
    for (int m = 0; m < width; m++)
    {
      int sum = 0;
      for (int i = -n2; i <= n2; i++)
      {
        for (int j = -m2; j <= m2; j++)
        {
          // Clamp result in bounds
          if (!outOfBounds(n - i, m - j, width, height))
          {
            sum += (K[i + n2][j + m2] * M[n - i][m - j]);
          }
        }
      }
    convolution[n][m] = sum;
    }
  }
  return convolution;
}
```

Computer Vision is the area of inferring properties given an image. Its pipeline can be simplified as



Relevant **kernels**

$$\underbrace{\underbrace{\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}}_{\text{Box}}\ \underbrace{\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}}_{\text{Gaussian}}}_{\text{Blur}}\quad \underbrace{\underbrace{\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}}_{\text{Laplacian}}}_{\text{Sharpening}}$$

$$\underbrace{\underbrace{\begin{bmatrix}i^2&0&1\\i^2&0&1\\i^2&0&1\end{bmatrix}}_{x}\ \underbrace{\begin{bmatrix}i^2&i^2&i^2\\0&0&0\\1&1&1\end{bmatrix}}_{y}}_{\text{Prewitt}}\quad \underbrace{\underbrace{\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}}_{x}\ \underbrace{\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}}_{y}}_{\text{Sobel}}$$

$$\text{Edge Detection}$$

CUDA: Parallel computing platform developed by NVIDIA. Stands for **C**ompute **U**nified **D**evice **A**rchitecture.

Before going into GPU, there are 5 steps for an instruction to finish (CPU-wise): (1) Fetch, (2) Decode, (3) Execute, (4) Memory Access, and (5) Register Write-Back. These steps are the five-stage for an **RISC** architecture.

One way of doing work in parallel is via **Instruction Level Parallelism** where in one clock cycle (of the CPU) many steps of different instructions are executed in parallel.

A CPU has a larger instruction set than a GPU, a complex ALU, a better branch prediction logic, and a more sophisticated caching/pipeline schemes. Instruction cycles are also faster.

The GPU comprises many cores processor and each core runs at a clock speed slower than a CPU's clock. GPU focus on execute throughput in parallel. For example, the GTX 280 GPU has 240 cores, each one multi-threaded, and with a SIMD (Single Instruction Multiple Data) paradigm, each core shares its control and instruction cache with the other seven cores.

Sequential programs do not have a "working set" of data, and most of its data can be stored in **L1**, **L2**, or **L3** cache, who are faster to fetch than from RAM.

Types of memories

- **DRAM** or Dynamic RAM. Slowest but least expensive (money-wise?).
- **SRAM** or Static RAM. Faster and does not require constant refreshing as DRAM. It is also known as **Cache Memory**. SRAM being expensive a processor would have few caches, for example, the Intel 486 microprocessor has only 8KB of SRAM.
- **VRAM** or Video RAM. Similar to DRAM but can be written-to and read-from simultaneously. An example usage is reading from VRAM and sending it directly to the screen without having to wait for the CPU to write into global memory.

The structure of a CUDA code has a GPU (device) part and a CPU (host) part. The host part is compiled by a traditional C compiler (like GCC), the device code requires a compiler that understands the special keywords used, an example of such compiler would be NVCC (NVIDIA C Compiler). When parsing the code, to differentiate what is host or device, NVCC looks for specific keywords, for example `__device__` or `kernel_name<<<>>>`.

Programmer has **explicit** control over the number of threads to be launched.

$$\text{threads} \xrightarrow{\text{packed into}} \text{blocks} \xrightarrow{\text{packed into}} \text{grids}$$

To execute a kernel on the GPU the usual pipeline is

1. Allocate memory on device (`cudaMalloc()`)
2. Transfer data from host memory into device memory (`cudaMemcpy()`)
3. Execute kernel (`kernel<<<blocksPerGrid,threadsPerBlock>>>()`)
4. Transfer data from device memory into host memory (`cudaMemcpy()`)
5. Release memory on device (`cudaFree()`)

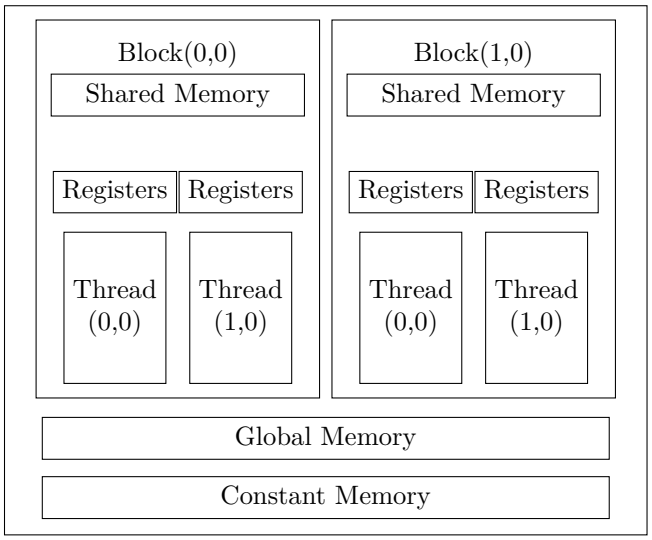
	Executed on	**Callable from**
`__device__`	GPU	CPU
`__global__`	CPU	GPU
`__host__`	GPU	GPU

Inside a kernel, important keywords to obtain indices are **gridDim**, **blockDim**, **blockIdx**, and **threadIdx**. Each one has 3 components `[x, y, z]`. For example, if we want to apply a kernel to an image of dimensions (76×62) we could launch $5120 = (80 \times 64)$, which is a multiple of 4, threads and each block having $256 = (16 \times 16)$ of them. Thus, a way of specifying the kernel is

```
dim3 threadsPerBlock(16,16,1)
dim3 blocksPerGrid(5,4,1)
kernel<<<blocksPerGrid,threadsPerBlock>>()
__device__ kernel()
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    ... remaining code ...
}
```

Execution resources assigned to threads per block are organized in **Streaming Multiprocessors** (SM). Multiple blocks of threads can be assigned to a single SM. After assigning to a SM, the block is divided into sets of 32 threads (called **warp**).

CUDA API has the built-in method `__syncthreads()` that causes all threads in a blocked to be blocked at the calling location until each thread reaches that point. This is to ensure **phase synchronization** when used.



Grid

- Global Memory: High-latency. Increase arithmetic in the program is to reduce accesses to global memory. All threads can access to it.
- Constant Memory: Short-latency. Total of 64KB on CUDA devices. Memory is cached. `__constant__` keyword is used to declare variables in this memory.
- Registers: On-chip memories. High-speed and concurrent. Kernel would store variables private to a thread into registers. SM 2.0 supports 63 registers while SM 3.5 expands to 255 registers.
- Shared Memory: Can be used to inter-thread communication. For accessing shared memory, processor needs to do a LOAD operation (registers do not require this).

Variable	**Memory**	**Scope**	**Lifetime**
Automatic (no array)	Register	Thread	Kernel
Automatic (array)	Local	Thread	Kernel
`__device__ __shared__`	Shared	Block	Kernel
`__device__`	Global	Grid	Application
`__device__ __constant__`	Constant	Grid	Application

Data Types

Keyword	Bytes [sizeof()]	Range
bool	1	[True,False]
unsigned char	1	$[0, 2^8)$
(signed) char	1	$[2^7 - 1, 2^7)$
unsigned int	4	$[0, 2^{32})$
(signed) int	4	$[2^{31} - 1, 2^{31})$
long	8	$[2^{63} - 1, 2^{63})$
float	4	Implementation-based
double	8	Implementation-based
void		

Operators

- Order of assignment is **left to right**, always `int x = y = z = 42;`
- Arithmetic `+, -, *, /, %`
- Bitwise `&, |, ∧, ∼, <<, >>`
- Logical `!, &&, ||`
- Relational `==, !=, >, <, >=, <=`
- Compound assignment `+=, -=, *=, /=, %=` `<<=, >>=, &=, |=`
- Pre-Post (In/De)crement

```
int x = 3;
int y = ++x; // y contains 4
int z = x++; // z contains 3
```

- Ternary `condition ? option1:option2;`
- Object size in bytes `sizeof()`

Statement and Flow Control

- Generic statements `{statement1; statement2; statement3;}`
- Control if `if (condition) statement;`
- Control while `while (expression) statement;`
- Control do-while `do {statement} while (condition);`
- For iteration `for (declaration : range) statement;`
- Jump statements `break; continue; goto label;`
- Selection

```
switch {expression}
{
    case constant1: statement; break;
    case constant2: statement; break;
    .
    .
    .
    default: statement; break;
}
```

Functions

`type name(parameter1, parameter2, ...){ statements; }`

- Parameters are passed **by value** (creates a copy) by default
- `const type &` to pass by reference
- Can be optimized (if small) using the `inline` keyword
- It can be made **recursive**
- Same name but different parameters types define **overloading**

```
void prototype(unsigned int x) {};
void prototype(float x) {};
int main()
{
    prototype(0); // Call unsigned int version
    prototype(0.0f); // Call float version
    return;
}
```

- Can use **templates** to maintain a method independent of data type

`template <typename T>`

`T function(const T a, const T b) { statements; }`

- To call a template function

`function <template_arguments>(function_arguments);`

```
// The following template is to avoid
// writing these 2 functions
// 1. int sum(int a, int b) { return a + b; }
// 2. float sum(float a, float b) { return a + b; }
template <typename T>
T sum(T a, T b) { return a + b; }
int main()
{
sum(1, 2) // Deduces to use int version of template
sum(1.0f, 2) // Will not compile due miss inferencing
sum<float>(1.0f, 2.0f) // Redundant infer but OK
return;
}
```

- Templates must be determined at **compile time**

```
template <class T, int N>
T multiply(T value) { return value * N; }
int main
{
multiply<int, 2>(10); // At compile time returns 20
multiply<int, 3>(10); // Similar but returning 30
return;
}
```

Scopes

- **static** storage ↦ global variable ↦ init to zero
- **automatic** storage ↦ local variable ↦ undetermined init
- outside any block ↦ global scope ↦ **global variable**
- within a block ↦ block scope ↦ **local variable**

```
int foo; // global variable
int do_something() { int foo; } // local variable
```

- Name collisions can be prevented with namespace keyword `namespace identifier { named_objects}`
- Access of namespace is with qualifier `::`

`namespace gg { int x }` ↦ `gg::x` access x

Arrays

- `type array_name[#elements]` ↦ declaration
- `array_name[array_index]` ↦ accessing
- Initialization does not need to be "complete"

```
// For an array of length N the C indexing is as follow
// Index   0   1   ...   N-1   N
// Array [  |   | ... |    |   ]
int foo[5] = {1, 2, 3, 4, 5} // [1, 2, 3, 4, 5]
int foo[5] = {1, 2, 3} // [1, 2, 3, 0, 0]
int foo[] = {1, 2, 3} // [1, 2, 3] Automatic size
int foo[3] = {} // [0, 0, 0]
```

- The `=` sign can be dropped for **universal initialization**
`int foo[]{10, 20, 30}` ↦

10	20	30

- Can be **multidimensional** `data[#rows][#cols]`, for example `data[3][2]` is represented as

	i = 0	i = 1
j = 2	$l^c_4 = 0$	$l^c_5 = 0$
j = 1	$l^c_2 = 0$	$l^c_3 = 0$
j = 0	$l^c_0 = 0$	$l^c_1 = 0$

	i = 0	i = 1
j = 2	$l^r_2 = 0$	$l^r_5 = 0$
j = 1	$l^r_1 = 0$	$l^r_4 = 0$
j = 0	$l^r_0 = 0$	$l^r_3 = 0$

```
// Elements of array data[3, 2] can be
// accessed/computed as
for (int i = 0; i < 2; i++)
for (int j = 0; j < 3; j++)
int linear_index_c = (i + j * 2); // Slow accessing?
int linear_index_r = (j + i * 3); // Fast accessing?
data[j][i] = 0; // Accessing with 2 indices
data[linear_index_c] = 0; // Accessing linearly - col
data[linear_index_r] = 0; // Accessing linearly - row
```

- As function parameters

```
// one-dimensional, size can be left blank
void procedure(int arg[]) {}
// multi-dimensional, 2nd or more dimensions requires
// size aka memory layout
void procedure(int array[][3]) {}
```

- An alternative for raw arrays in `C++` is the container `std::array<type, size>`, which can be accessed via the `#include <array>` header

Pointers

- `type *` represents a **pointer** to a type
- `&` is the **reference** operator and can be read as "address of"
- `*` is the **dereference** operator and can be read as "value pointed by"

```
int var = 25; // Initial variable var with value 25
int * foo = &var; // foo points to the address of var
int bar = var; // bar is a new variable of value var
int baz = *foo; // baz dereference foo, gets 25
```

- Pointers can be manipulated "arithmetically"

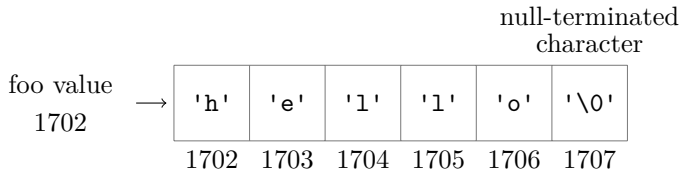
```
void console_log_p(int * p) { printf("%p:%d, ", p, *p);
    }
int main()
{
    int numbers[5];
    int * p;
    p = numbers; *p = 10; console_log_p(p);
    p++; *p = 20; console_log_p(p);
    p = &numbers[2]; *p = 30; console_log_p(p);
    p = (numbers + 3); *p = 40; console_log_p(p);
    p = numbers; *(p + 4) = 50; console_log_p(p);
    return;
}
```

Suppose `0x567` is the address of `p`, above code prints
`0x567:10, 0x56A:20, 0x56E:30, 0x572:40, 0x576:50`

- "Constantsy" of pointers

```
int x; // non-const int variable
int* p1 = &x; // non-const pointer to non-const int
const int* p2 = &x; // non-const pointer to const int
int* const p3 = &x; // const pointer to non-const int
const int* const p4 = &x; // const pointer to const int
```

- String literals ("text") can be represented as
`const char * foo = "hello"`



- A pointer can point to a pointer

```
char a; a = 'z'; // a has value of character z
char * b; b = &a; // b points to a
char ** c; c = &b; // c points to b
```

- Initialization `int * p = 0; int * q = nullptr;`
`nullptr` ↦ points to "nowhere"
`void *` ↦ can point to somewhere
- Pointer to function
`type (*name)(parameters) = function_name;`

Dynamic Memory

- Uses keyword `new` to allocate memory as in
`pointer = new type` or `pointer = new type[#elements]`
- To release memory use `delete` or `delete[]` keyword

```
int * poo = new int; // Allocate an int
*poo = 1; // Populate with 1
int * foo = new int[5] // An array of 5 elements
int elements = 3;
int * hoo = new int[elements] // An array of 3 elements
// Clear memory in reverse order
delete[] hoo;
delete[] foo;
delete poo;
```

- `C` can handle dynamic memory using `malloc`, `calloc`, `realloc` functions but requires the `#include<cstdlib>` header

Data Structures

- `struct type_name`
  `{ member_type_n member_name_n; } object;`
- Accessing of members via the `.` operator

```
struct data { int x; int y; int z; } coordinates;
coordinates.x = 1; // Assigning 1 to member x
// Assigning members y and z
coordinates.y = coordinates.z = 0;
```

- Use the `→` to access the members of a pointer to struct. Alternatively, one can dereference and then access a member, i.e. `(*pointer_to_struct).`

```
struct fruits_t
{ std::string apple; std::string banana; };
fruits_t afruit;
fruits_t * pfruit = &afruit;
pfruit->apple = "apple"; // Access via -> operator
// Access via dereference-member, i.e. (*).
(*pfruit).banana = "banana";
```

- `class` keyword is similar to struct (in holding variables) but with different default access to members
  `struct` ↦ default **public** access
  `class` ↦ default **private** access

```
class foo_ct { public: int x; } foo_c;
struct foo_st { int x; } foo_s;
foo_c.x = 0; // Accessible with public: modifier
foo_s.x = 0; // Accessible by default
```

- Another keyword similar to struct is `union`. While struct (and class) allocates memory for each of its members, union will allocate only a chunk to be reused among members

```
union mix_t {
    int l; struct { short hi; short lo; }; char c[4];
} mix_example;
printf("%d\n", sizeof(mix_example)); // Prints 4
```

Enumerated Types

- `enum class type_name : data_type`
  `{ value1 = init_val, value2, ... } object_names;`
- Default underlying memory is `int`

```
enum ColorEnumDefault { red = 42, green, blue };
enum class ColorEnum : char { red, green, blue };
printf("%d\n", red); // OK but easily to do name clash
printf("%d\n", green); // Prints 43
// Prints 4 and better intent for access
printf("%d\n", sizeof(ColorEnumDefault::red));
// Prints 1 and no ambiguity
printf("%d\n", sizeof(ColorEnum::red));
printf("%d\n", ColorEnum::green); // Prints 1
```

Aliases

- Two keywords, `typedef` or `using`
  `typedef existing_type new_type_name;`
  `using new_type_name = existing_type;`

```
typedef unsigned char uchar; using uint = unsigned int;
uchar a = 0; // OK to use
uint b = 1; // Also OK to use
```

Classes

- `class class_name`
  `{ access_n : type_m member_name_m; } object;`
- Creation of object ↦ Constructor `(), =, {}`
  Destruction of object ↦ Destructor `~`
- Keyword `this` is a pointer to the object whose member functions is being executed

```
class Rectangle
{
    int w, h;
public:
    void log_values() { printf("w:%d, h:%d\n", w, h); }
    void set(int, int);
    // This function could be inlined
    int area() { return w * h; }
};
void Rectangle::set(int _w, int _h)
{
    // w, h are private but accessible in class scope
    // *this* keyword represents members of the class
    this->w = _w;
    h = _h;
}
// OK, call default constructor but might init garbage
Rectangle r_1;
r_1.log_values();
// Warning, r_2() is calling a function not constructor
Rectangle r_2();
// Error, r_2 has not been created as Rectangle
// r_2.log_values();
// Also default constructor but init to zero
Rectangle r_3{};
r_3.log_values();
```

- Class instantiation can be done in multiple forms : **default or functional**, **assignment**, **copy**. **move**, **list**

```
class Circle
{
    double r;
public:
    void log_values() { printf("%f\n", r); }
    Circle(double _r) { r = _r; }
    Circle& operator=(const Circle& c)
    { r = c.r; return *this; }
};
// Default construct is not available
// due to defining one constructor with a parameter
Circle foo(10.0); // Functional form
Circle bar = 20.0; // Default assignment
Circle but = bar; // Defined assignment?
Circle baz{ 30.0 }; // Uniform init
Circle qux = { 40.0 }; // ?
foo.log_values(); // 10
bar.log_values(); but.log_values(); // 20
baz.log_values(); qux.log_values(); // 30, 40
```

- `static` keyword in a class acts like a "global" variable that requires class scope for accessing
- A static member function of a class cannot access non-static members

```
class Dummy
{
    int m; // Normal member
public:
    static int n; // Requires definition in global space
    static void log_s()
    {
        //printf("m : %d\n", m); Cannot access m member
        //log(); Cannot access log function
        printf("n : %d\n", Dummy::n);
    }
    void log_c() { printf("n, m : %d, %d\n", n, m); }
    Dummy(int _m) { m = _m; n++; } ~Dummy() { n -= 2; }
};
int Dummy::n = 0;
Dummy::log_s(); // Prints 0
Dummy dummy1(0); // n increments 1
dummy1.log_c(); // Prints m=0, n=1
Dummy::log_s(); // Prints 1
{
    Dummy dummy2(1); // n increments 1
    dummy2.log_c(); // Prints m=1, n=2
    Dummy::log_s(); // Prints 2
} // Finish scope, dummy2 calls destructor
dummy1.log_c(); // Prints m=0, n=0
Dummy::log_s(); // Prints 0
```

- `const` blocks the availability to change values, either by instantiation or by getting values

```
class MyClass
{
public:
    int x;
    MyClass(int _x) : x(_x) {};
    int get_nonconst() { return x; }
    // x++; Not allow to modify content in function
    int get_const1() const { return x; }
    // Allow to modify, but returns a const number
    const int get_const2()
    {
        x++; const int y = x;
        return y;
    }
};
const MyClass MyFoo(10);
MyClass MyBar(20);
// MyFoo.x = 20; Not valid, MyFoo is const
// MyFoo.get_nonconst(); Not valid, non const method
// MyFoo.get_const2(); Also not valid
int x = MyFoo.get_const1(); // 10
int y = MyBar.get_nonconst(); // 20
int z = MyBar.get_const1(); // 20
int w = MyBar.get_const2(); // 21
```

Possible options are

```
// const member function
int get() const { return x; }
// member function returning const &
const int & get() { return x; }
// const member function returning const &
const int & get() const { return x; }
```

- Suppose `C` a class

Class special member	Syntax
Default Constructor	`C::C();`
Destructor	`C::~C();`
Copy Constructor	`C::C(const C&);`
Copy Assignment	`C& operator=(const C&);`
Move Constructor	`C::C(C&&);`
Move Assignment	`C& operator=(C&&);`

```
class C {
private: int x = 0;
public:
    C() { printf("Constructor C\n"); }
    ~C() { printf("Destructor C\n"); }
    C(const C&) { printf("Copy Constructor C\n"); }
    C& operator=(const C&)
    { printf("Copy Assignment C\n"); }
    C(C&&) { printf("Move Constructor C\n"); }
    C& operator=(C&&) { printf("Move Assignment"); }
    void log(char s)
    { printf("%c : %p -> %d(%p)\n", s, this, x, &x); }
};
printf("%d\n", sizeof(C));
C objectC_a = C();
objectC_a.log('A');
C objectC_b = C(objectC_a);
objectC_b.log('B');
objectC_a.log('A');
C objectC_c = C(std::move(objectC_a));
objectC_c.log('C');
objectC_b.log('B');
objectC_a.log('A');
```

Class Templates

- Declaration ↦ `template <class T> class class_name{ };`
  Instantiation ↦ `class_name<T> object_name( );`

```
template <class T> class Pair {
    T values[2];
public:
    Pair(T a, T b) { values[0] = a; values[1] = b; }
    void log() {
        printf("%d bytes\n", sizeof(T));
        printf("[0] %p->%f\n", &values[0], values[0]);
        printf("[1] %p->%f\n", &values[1], values[1]);
    }
};
// Creates a Pair float, Pair double, respectively
Pair<float> pairFloat(1.0f, 2.0f); pairFloat.log();
Pair<double> pairDouble(3.0, 4.0); pairDouble.log();
```

- Can be specialized for specific type of data types

```
template <typename T> class Foo {
    public: Foo(T arg)
    { printf("Generic template : %s\n", arg.c_str()); }
};
template <> class Foo<char> {
    public: Foo(char arg)
    { printf("Specialized template : %c\n", arg); }
};
Foo<std::string>(std::string("hello std::string"));
Foo<char>('A');
```

Inheritance

- `class derived_class : qualifier base_class_name { };`
- Base class can be an **interface** using the `virtual` keyword and the `= 0` post-declaration to get **pure virtual** behavior
- The `qualifier` can be `public`, `protected`, or `private`

Access	public	protected	private
base members	Y	Y	Y
derived members	Y	Y	N
not members	Y	N	N

```
namespace MX
{
    class Polygon
    {
    protected:
        int w = 0, h = 0;
    public:
        Polygon() {}
        ~Polygon() { printf("Destructor Polygon\n"); }
        Polygon(int w, int h) : w(w), h(h) {}
        virtual float area() = 0; // pure virtual function
        void log() { printf("Area : %f\n", this->area()); }
    };
    class Rectangle : public Polygon {
    public:
        Rectangle(int w = 1, int h = 1) : Polygon(w, h) {}
        ~Rectangle() { printf("Destructor Rectangle\n"); }
        float area() { return float(w * h); }
    };
    class Triangle : public Polygon {
    public:
        Triangle(int w = 1, int h = 1) : Polygon(w, h) {}
        float area() { return (w * h * 0.5f); }
    };
    class Pentagon : public Polygon {
    public:
        Pentagon(int side = 1) {
            // TODO: Compute w,h for a pentagon using side
            this->w = 0; this->h = 0; }
    };
}
// auto polygon = MX::Polygon(0, 0); Error, Pure virtual
// Polymorphism (* base_class = * derived_class)
MX::Polygon * rectangle1 = new MX::Rectangle();
rectangle1->log();
delete rectangle1; // Call to MX::~Polygon()
{ auto rectangle2 = MX::Rectangle(2, 2);
// Call to MX::~Rectangle and then MX::~Polygon()
rectangle2.log(); }
auto triangle = MX::Triangle();
// Call to MX::~Polygon, MX::~Triangle() not defined
triangle.log();
// MX::Pentagon(); Error, area() does not have overrider
```

STL (Standard Template Library) Containers

- A **container** is an object that **stores a collection** of other objects
- The container **manages the storage space** for its elements and gives member **functions to access** them
- STL uses the namespace `std`
- Can be subdivided in **four categories**

Sequence Containers Data structures that can be accessed sequentially	
`array`	Static contiguous array
`vector`	Dynamic contiguous array
`deque`	Double-ended queue
`forward_list`	Singly-linked list
`list`	Doubly-linked list
Average search requires $O(n)$	

Ordered Associative Containers Data structure that is kept sorted (by keys)	
`set`	Collection of unique keys
`map`	Collection of key-value pairs with unique keys
`multiset`	Collection of keys
`multimap`	Collection of key-value pairs
Average search requires $O(\log n)$	

Unordered Associative Containers Unsorted (hashed by keys) data structure	
`unordered_set`	Collection of unique keys
`unordered_map`	Collection of key-value pairs with unique keys
`unordered_multiset`	Collection of keys
`unordered_multimap`	Collection of key-value pairs
Average search requires $O(1)$ amortized **Worst case search** requires $O(n)$ amortized	

Container adaptors Different interface for sequential containers	
`stack`	LIFO data structure
`queue`	FIFO data structure
`priority_queue`	First element is the greatest and elements are in nonincreasing order
Average search requires $O(n)$	