FUNDAMENTOS DE PROGRAMAÇÃO DE JOGOS

conceitos básicos

CONTEÚDO

01. LÓGICA DE PROGR...

Lógica fundamental aplicada à programação

02. DES... ALGORITMOS

Habilidades de design e análise de algoritmos

03. ESTRUTURAS DE DADOS

Peças de lego da programação *(Building blocks)*

04. FUNDAMENTOS DE PRO...

Conceitos de linguagens e paradigmas de programação

05. INTUIÇ... MATEMÁTICA

Como a matemática se relaciona com jogos

01. LÓGICA DE PROGRAMAÇÃO

Lógica fundamental aplicada a programação

LÓGICA DE PROGRAMAÇÃO

ÁLGEBRA DE BOOLE

A álgebra booliana descreve as propriedades fundamentais da lógica.

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\neg$	0	1
	1	0

Por meio de símbolos matemáticos podemos “operar” sobre expressões que podem ser usadas em conjunturas argumentativas mais elaboradas.

Na programação, ela descreve operações fundamentais que podem ser feitas com o sistema numérico binário.

A lógica na programação também pode ser estendida para sistemas mais complexos como, por exemplo, inteligências artificiais.

ÁLGEBRA DE BOOLE

OPERADORES

Conectivo	Símbolo	Operação Lógica	Valor Lógico
não	~	negação	Terá valor falso quando a proposição for verdadeira e vice-versa.
e	^	conjunção	Será verdadeira somente quando todas as proposições forem verdadeiras.
ou	v	disjunção	Será verdadeira quando pelo menos uma das proposições for verdadeira.
se...então	→	condicional	Será falsa quando a proposição antecedente for verdadeira e a consequente for falsa.
...se somente se...	↔	bicondicional	Será verdadeira quando ambas as proposições forem verdadeira ou ambas falsas.

LÓGICA DE PROGRAMAÇÃO

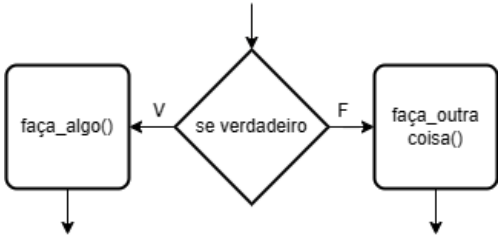
INFERÊNCIA

Inferência é uma das ferramentas mais importantes da lógica. É fundamentalmente descrita pela operação condicional, e é por meio dela que podemos tomar conclusões indiretas em um conjunto de dados.

Na programação de jogos, usamos inferência a todo momento na forma das estruturas condicionais if-else.

TL;DR: Usamos inferência (condicional) para chegar a resultados distintos conforme algo é verdadeiro ou falso.

```
if true:
»   faça_algo()
else:
»   faça_outra_coisa()
```



02.

DESENVOLVIMENTO DE ALGORITMOS

Habilidades de design e análise

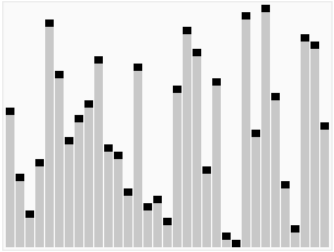
DESENVOLVIMENTO DE ALGORITMOS

O QUE É UM ALGORITMO

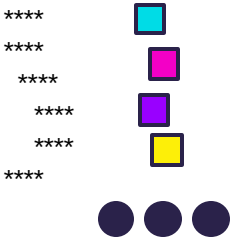
Em matemática, e na ciência da computação, um algoritmo é descrito formalmente como “uma sequência de **passos finitos** de **ações** executáveis que visam obter uma **solução** para um dado **problema**”.

Em termos práticos, algoritmo é o código resultante de uma funcionalidade escrita por um programador.

É comum estruturarmos algoritmos como uma sequência de **expressões** - uma após outra - que são separadas em **blocos**, delimitados por endentação (recuo). Note que blocos de código podem conter outros sub-blocos.



Algoritmo quicksort



LÓGICA DE PROGRAMAÇÃO

TABELAS VERDADE

Tabelas verdade é uma “ferramenta” usada para verificar as propriedades de uma expressão lógica e sua veracidade.

Quando todas as células de uma coluna da tabela verdade são verdadeiros (1), dizemos que temos uma **tautologia**. Em contraposição, se todos forem falsos (0) teremos uma **contradição**.

Dessa forma podemos analisar conjunturas lógicas para aprimorar argumentos ou circuitos e reduzir complexidade, ou chegar em novas conclusões.

Negação (~)

A	~A
V	F
F	V

Conjunção (∧)

A	B	A ∧ B
V	V	V
V	F	F
F	V	F
F	F	F

Disjunção (v)

A	B	AvB
V	V	V
V	F	V
F	V	V
F	F	F

Condicional

A	B	A→B
V	V	V
V	F	F
F	V	V
F	F	V

Bicondicional

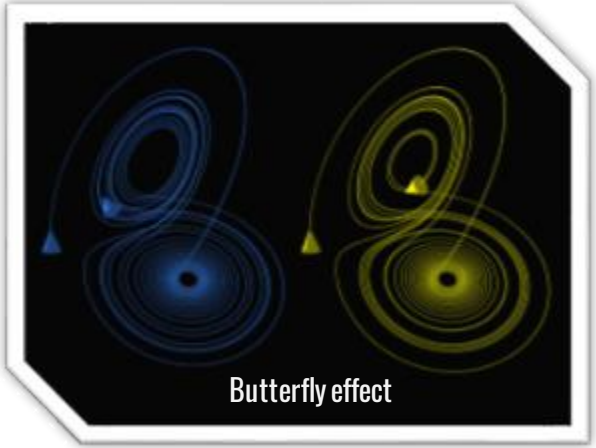
A	B	A↔B
V	V	V
V	F	F
F	V	F
F	F	V

DESENVOLVIMENTO DE ALGORITMOS

ESTRUTURAS

De forma conceitual, há duas estruturas básicas em algoritmos:

- As estruturas **condicionais**:
  Determinam caminhos (blocos de código) diferentes para serem executadas conforme alguma expressão lógica for atendida.
  Ex.: o famoso if-else



- As estruturas de **repetição**:
  Executam blocos de código repetidas vezes, conforme **variáveis** vão mudando ao longo da execução.
  Ex.: os loops (que "idealmente" não são infinitos)

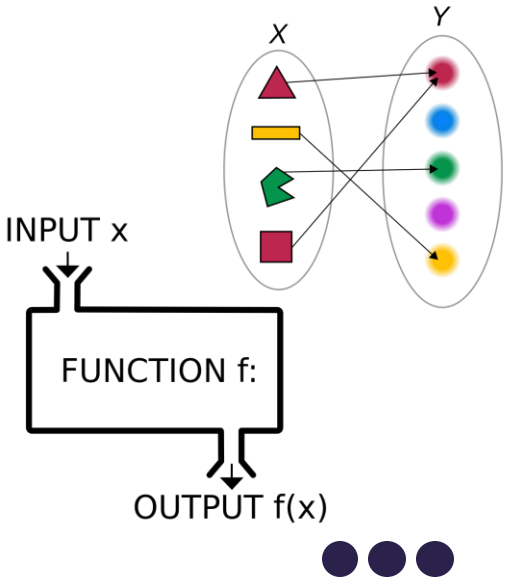
DESENVOLVIMENTO DE ALGORITMOS

FUNÇÃO

Na matemática, uma função é descrita como uma **relação** entre elementos de **conjuntos**. Ex.: X -> Y, f(x) ∈ Y, x ∈ X

Na programação uma função é, em termos práticos, um bloco de código que descreve parte de um algoritmo. De forma que esse algoritmo possa ser reutilizado diversas vezes conforme essa função é **chamada** por meio de uma expressão simples.

Essa expressão, imitando a forma matemática, recebe uma lista de valores (**entrada**) entre parênteses e retorna um valor (**saída**) para ser usada em expressões compostas.
Ex.: função(argumento1, argumento2, ...)



DESENVOLVIMENTO DE ALGORITMOS

RESOLUÇÃO DE PROBLEMAS

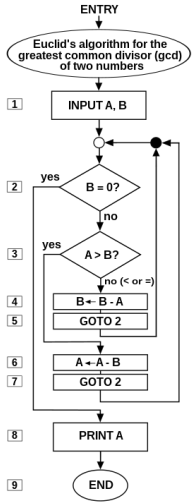
O principal objetivo dos algoritmos é resolver **problemas** por meio da computação (processamento de um conjunto de dados e execução de tarefas). De forma que dado uma mesma entrada, obtemos a mesma saída (algoritmos **determinísticos**), ou uma saída prevista e satisfatória (algoritmos probabilísticos ou **aleatórios**).

Algoritmos devem ser **corretos** (atinge a solução esperada), podendo ser **exatos** ou **aproximados**, **eficientes** (rápidos e econômicos) ou não. Os passos típicos para desenvolvimento de algoritmos são os seguintes:

1. Definição do problema
2. Desenvolvimento do modelo
3. Especificação do algoritmo
4. Design do algoritmo
5. Verificar sua corretude
6. Análise do algoritmo
7. Teste da solução num programa de software
8. Documentação

1599

Euclid's algorithm finding the greatest common divisor for 1599 and 650



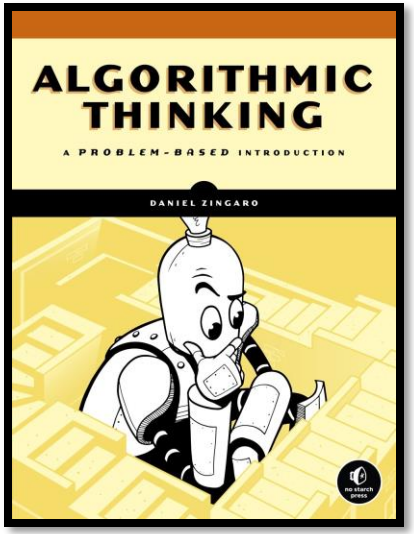
DESENVOLVIMENTO DE ALGORITMOS

PENSAMENTO ALGORÍTMICO

Programadores experientes desenvolvem a **intuição** para resolução de problemas complexos, fundada na **prática** em resolver algoritmos pequenos e **abstração** das formas.

Não é necessário resolver problemas que já foram resolvidos.
Estuda-se algoritmos para desenvolver esta intuição. Assim, alcançar soluções inovadoras, remodelar soluções antigas em formas mais eficientes, ou simplesmente aplicar as soluções existentes em novos produtos de software.

Jogos digitais, em especial, são sistemas complexos que utilizam de toda a amplitude de áreas da Ciência da Computação como ferramenta. - Mas nós, programadores, precisamos apenas nos ater a como usar estas ferramentas ao nosso favor, nos abstraindo de toda a complexidade da essência dos problemas envolvidos.



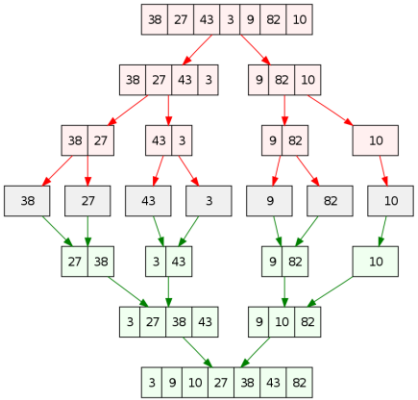
DESENVOLVIMENTO DE ALGORITMOS

PROJETO & DESIGN DE ALGORITMOS

Visto isso, programadores desenvolveram ao longo da história diversas técnicas para construção desses algoritmos, com o objetivo de compreender e resolver os problemas da melhor forma.

Exemplos de técnicas de design:

- **Força bruta** ou busca exaustiva: é uma técnica, em geral ineficiente, para resolver problemas de forma direta, que consiste em tentar cada solução possível até encontrar uma válida.
- **Dividir para conquistar**: divide-se o problema em subproblemas cada vez menores de forma que a solução seja construída em partes.
- **Busca e enumeração**: obtemos soluções por meio da exploração - de forma bem direcionada - e medida de um espaço de dados.
- **Algoritmos randomizados**: muitas vezes é pertinente para o programador fazer escolhas de forma aleatória de forma a obter soluções aproximadas que outrora seriam muito difíceis de serem alcançadas.
- **Transformar para conquistar** / Redução de complexidade: para obter soluções melhores, é comum que programadores transformem um conjunto de dados em outros modificando os meios de resolução.
- **Backtracking** / Retrocesso: é uma técnica que envolve a construção de uma série de candidatos para resolver um problema e abandoná-los logo que for determinado que não produzem uma solução válida.



divide & conquer
to sort a list

03. ESTRUTURAS DE DADOS

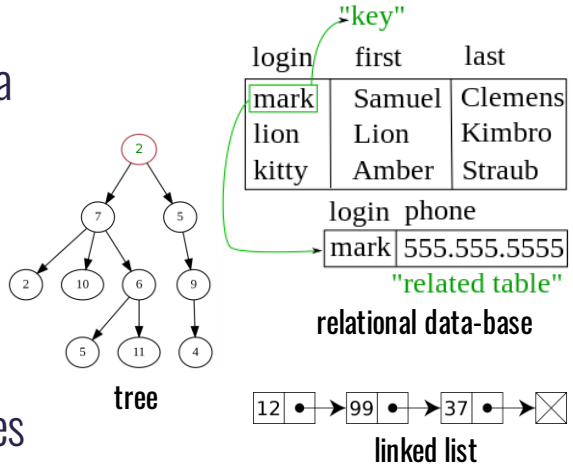
Peças de lego da programação

ESTRUTURAS DE DADOS

O QUE É UMA ESTRUTURA DE DADOS

Na Ciência da Computação, uma estrutura de dados é uma forma de organizar, gerenciar e armazenar dados – em geral, escolhida pelo programador para tirar proveito de métodos eficientes de acesso a esses dados.

Mais precisamente, é uma coleção de valores, o relacionamento entre eles e a série de funções e operações que podem ser feitas sobre eles.



ESTRUTURAS DE DADOS

VARIÁVEL

Na matemática, podemos definir variáveis formalmente como o conjunto de todos os valores possíveis para algo.
Este "algo" é o que caracteriza a variável.

Quando fixamos o valor de uma variável, em uma dada observação, dizemos que esse valor é uma **instância** daquele conjunto.

O conjunto de valores de uma variável também determina seu **tipo**.
Ex.: $\{x \mid x \in Z\}$, dizemos que x é do tipo inteiro.

variáveis na matemática

$$y = x$$

$$y = bx + ax^0$$

$$y = cx^2 + bx^1 + ax^0$$

$$y = a_0x^n + a_1x^n + \cdots a_{n-1}x^1 + anx^0$$

$$y = \sum_{i=0}^{n} a_{n-i}x^i$$

$$y = f(x)$$

VARIÁVEL

VARIÁVEIS NA PROGRAMAÇÃO

Na programação, determinamos o tipo de uma variável de forma implícita ou explícita.

```
var x      # x é uma variável de tipo "variante"
var x: int # x é uma variável de tipo inteiro
```

Tecnicamente, uma variável em programas de computador é apenas uma sequência binária.
Ex.: $x = 0101100101$

Mas, nós programadores podemos nos abstrair desse fato considerando apenas o significado da variável. Ex.: $\pi \in \{x \mid x \in \mathbb{R}\}$ π é um dado de tipo real.

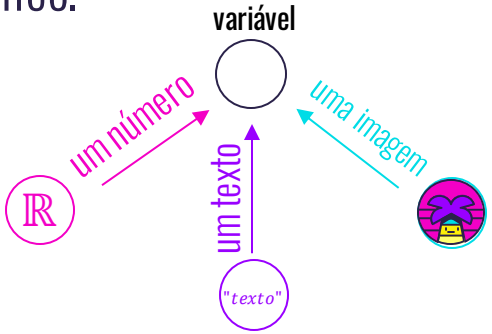
- Note que o computador não é capaz de armazenar o valor real de π dado que não há capacidade de memória infinita para tal.

VARIÁVEL

VARIÁVEIS E ESTRUTURAS DE DADOS

Visto isso, vamos definir a ideia de variável como um “**espaço**” na memória do computador que armazena uma certa **instância** de um **tipo** específico.

TL;DR:



ESTRUTURAS DE DADOS

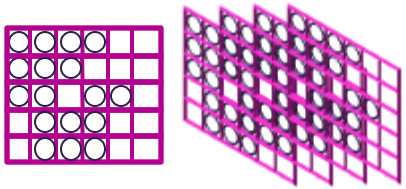
ESTRUTURAS PRIMÁRIAS

Variáveis podem ser agrupadas de forma a construir estruturas mais complexas:

Uma estrutura composta pode ser:

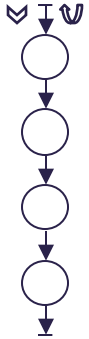
- Vetores (array)

- Matrizes



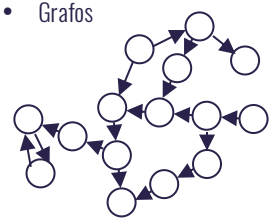
- Filas

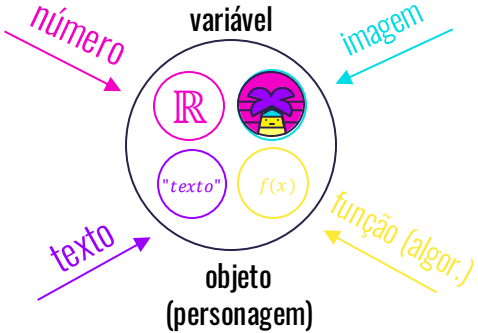
- Pilhas



- Árvores

- Grafos





ESTRUTURAS DE DADOS

VARIÁVEIS COMPOSTAS

Variáveis podem ser simples (armazenam um único valor), ou compostas (armazenam múltiplos valores).
Variáveis simples são também chamadas de **atômicas** - exemplos: booleanos (V, F), inteiros, reais, caracteres, endereços.

Exemplos de variáveis compostas:

- **String** (cadeia de caracteres): "texto", "olá", "🤓", "a", "\n", ""
- **Vetor** algébrico: (x, y), (x, y, z), (2.5, 1.14, 17),
- **Cor**: (r, g, b), (l, h s), (255, 255, 255), #fffff

Na programação orientada a objetos, chamamos uma estrutura composta de outras e que também possuem comportamentos (funções / algoritmos) de **objetos**. Exemplos:

- Array: A
- Matriz: M
- Árvore: T
- Grafo: G

```
A[0]    # Acessa o primeiro elemento do array
M[1][0] # Acessa o elemento na primeira coluna e segunda linha
T.insert(nó) # Insere o nó na árvore
G.find("nó") # Busca um "nó" no grafo
```

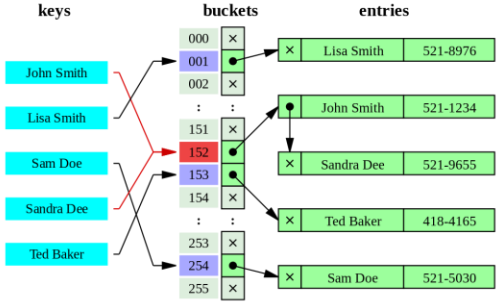
ESTRUTURAS DE DADOS

REPRESENTAÇÃO DE DADOS

Estruturas de dados podem ser representadas de diversas maneiras, seja logicamente (no papel) ou fisicamente (na memória do computador). O programador, é claro, escolhe aquelas mais convenientes para solucionar o seu problema.

No geral, toda estrutura de dados pode ser abstraída ao máximo como um grafo.
Grafo > Árvore > Lista (pode ser Pilha ou Fila) > Variável

E todo grafo pode ser representado como uma matriz.
Matriz (Grafo, Tabela ou Relação) > Vetor > Variável



hash table

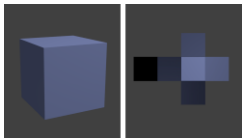
Na matemática, grafos e matrizes são definidos por funções (relações).
Existe um tipo especial de estrutura de dados que permite associar diretamente um valor (“chave”) a outro: as **tabelas de dispersão** (hash table). Funcionam como uma espécie de dicionário que “traduz” um valor em outro.

ESTRUTURAS DE DADOS

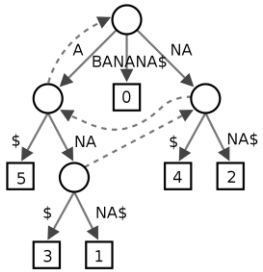
COMBINAÇÕES DE ESTRUTURAS

É possível combinar e representar estruturas de dados de diversas maneiras, em grandes magnitudes de complexidade.

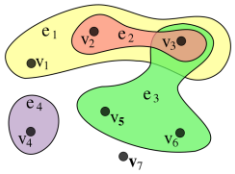
Dominar as estruturas de dados torna o programador capaz de resolver uma grande gama de problemas computacionais.



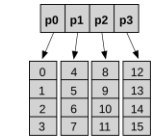
light map



suffix tree

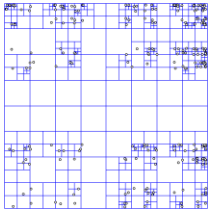


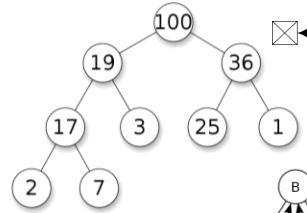
hyper graph



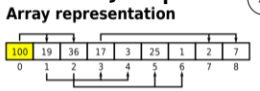
hashed array tree

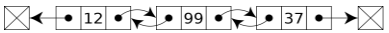
quad tree



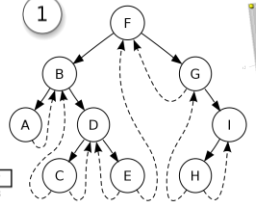


binary heap

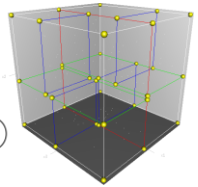




doubly linked list



threaded tree



k-d tree

04. FUNDAMENTOS DE PROGRAMAÇÃO

Conceitos de linguagens &
paradigmas de programação

FUNDAMENTOS DE PROGRAMAÇÃO

SINTAXE & SEMÂNTICA

Na linguística, sintaxe é o estudo de como as palavras e morfemas se combinam para formar **unidades maiores**, como sentenças e frases.
De forma similar, na computação, a sintaxe se refere a como uma linguagem de programação deve estar **estruturada** para ser considerada um código válido (executável).

Já a semântica, que deriva do Grego sēmantikós, "significante", é o estudo das referências, significados ou da verdade.
Estudando o sentido das unidades linguísticas do discurso (palavras, sentenças e frases), analisando a **composição** e o **contexto** dessas unidades.
Na programação, se refere ao **significado das construções** da linguagem, ao invés de suas formas como abordado pela sintaxe.

TL;DR: Sintaxe = forma & estrutura. Semântica = significado & contexto.

SINTAXE & SEMÂNTICA

EXEMPLO DE SINTAXE

A linguagem de programação C possui em sua especificação, regras objetivas e minuciosas de como um código deve ser estruturado. Um programa escrito por um programador C é valido apenas se tais restrições forem seguidas à risca. Diferente da linguística que admite alguns erros sem prejudicar o significado passado ao intérprete.

```c
long some_function(); /* This is a function declaration, so the compiler can know the name and return type of this function. */
/* int */ other_function(); /* Another function declaration.  There is an implicit 'int' type here since we're talking about early version of C.  It's commented out here to show where it could go in later variants. */

/* int */ calling_function() /* this is a function definition, including the body of the code following in the { curly brackets } the return type is 'int', but this is implicit so no need to state 'int' when using this early version of C */
{
    long test1;
    register /* int */ test2; /* again, note the 'int' is not required here, and shown as */
                              /* a comment just to illustrate where it would be required in later variants of C. */
                              /* The 'register' keyword indicates to the compiler that this variable should */
                              /* ideally be stored in a register as opposed to within the stack frame. */
    test1 = some_function();
    if (test1 > 1)
          test2 = 0;
    else
          test2 = other_function();
    return test2;
}
```

SINTAXE & SEMÂNTICA

EXEMPLO DE SEMÂNTICA

Embora um programa siga as regras da linguagem à risca, ainda é possível ocorrer falhas no decorrer de sua execução. Isso é devido ao que é popularmente chamado de “erros de lógica”, que na verdade são erros semânticos – o significado que o programador buscava atingir é diferente daquilo que está escrito, em geral, o significado do código apresenta obstáculos que não foram previstos pelo programador durante sua construção.

```
Enter an integer: 5
5 is greater than 5
```

```cpp
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    if (x >= 5) // oops, we used operator>= instead of operator>
        std::cout << x << " is greater than 5\n";

    return 0;
}
```

FUNDAMENTOS DE PROGRAMAÇÃO

DETECÇÃO DE PADRÕES

Como foi dito na seção de desenvolvimento de algoritmos, programadores desenvolvem ao longo de sua carreira habilidades de intuição para resolução de problemas. Parte desta intuição está enraizada na habilidade de detecção de padrões.

Assim como certas estruturas de dados, técnicas de design de algoritmos e formulações matemáticas se relacionam com certos tipos de problemas; as linguagens de programação, ou mais especificamente os paradigmas destas linguagens, projetam uma "aura" ou forma que o programador pode associar a uma gama de diferentes soluções.

Aqui, analisamos as formas de pensar de um programador nos tópicos listados à seguir:

- Modelagem lógica e estrutural;
- Interpretação de dados;
- Design e arquitetura de software;
- Intuição matemática, geométrica e física.

FUNDAMENTOS DE PROGRAMAÇÃO

DETECÇÃO DE PADRÕES

LÓGICA & ESTRUTURA

Um programador experiente pensa de forma assertiva, estruturando suas ideias de forma significativa e cada vez mais complexas.

DESIGN & ARQUITETURA

Um bom programador é acima de tudo um criador. A experiência e a imaginação tem um papel essencial na construção da intuição.

INTERPRETAÇÃO DE DADOS

Um bom programador sabe identificar e direcionar os caminhos para processar as informações (I/O), resignificando tais dados.

INTUIÇÃO MATEMÁTICA

A matemática e a geometria são ferramentas da abstração que permitem engenheiros e físicos “dominarem” o mundo.

FUNDAMENTOS DE PROGRAMAÇÃO

PARADIGMAS DE PROGRAMAÇÃO

Os paradigmas de programação são formas de classificar as linguagens de programação. Assim como uma pessoa que fala uma certa linguagem pode moldar uma nova personalidade ao aprender uma segunda[x], os paradigmas pregam um papel essencial nas formas de pensar do programador.

Alguns paradigmas estão interessados nas implicações dos modelos de execução – como, por exemplo, permitir efeitos colaterais - enquanto outros observam como uma base de código é organizada – como a divisão de grandes corpos de código em unidades, seus relacionamentos e restrições sobre o estado e modificações – já outros voltam-se ao estilo, sintaxe ou gramática da linguagem.
Alguns paradigmas de programação:

- Imperativos: procedural ou orientado a objetos.
- Declarativos: funcional, lógico, matemático, reativo, restritivos (constraint - CP), linguagens de query...
- Modelos: programação concorrente, design orientada a dados, simbólica, e meta-programação.

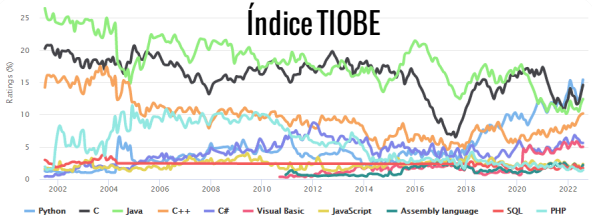
PARADIGMAS

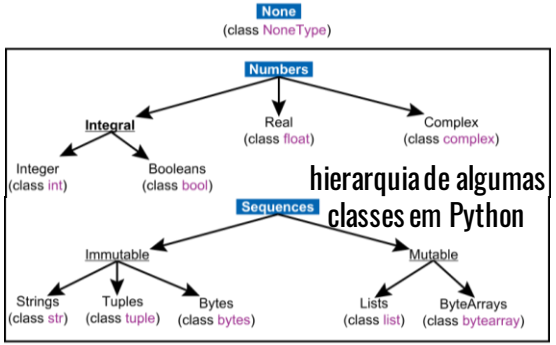
FORMAS DE PENSAR DO PROGRAMADOR

Dentre os paradigmas listados, os mais usados são os imperativos (procedural e orientado a objetos).

A orientação à objetos é uma reorganização do modelo procedural, que divide os segmentos (blocos) de código em **classes**, e pensa nas unidades (instâncias) que compõem o programa como **objetos**.

As linguagens modernas, muitas vezes, não se restringem a um único paradigma e podem envolver conceitos de diversos outros.
A linguagem Python, por exemplo é inerentemente orientada a objetos, mas utiliza-se de práticas de meta-programação e programação funcional em seu pacote de features (características ou capacidades).





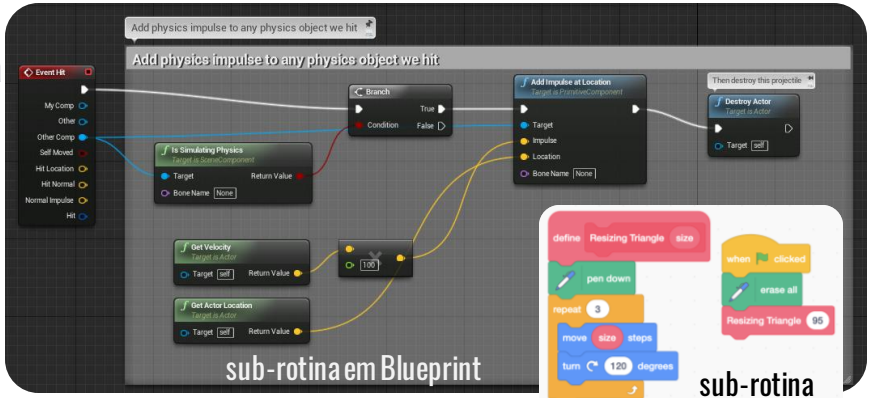
PARADIGMAS

PROGRAMAÇÃO PROCEDURAL

É um paradigma imperativo baseado no conceito de chamada procedural. Um procedure (um tipo de rotina ou sub-rotina) nada mais é uma série de passos computacionais que podem ser executados um-a-um a qualquer momento durante a execução de um programa sempre que for feita uma **chamada** a essa procedure.

Procedures, inspiradas nas funções, podem receber uma série de variáveis de entrada e retornar uma saída; mas, além disso, podem operar sobre variáveis externas a elas. Chamamos este conceito de efeitos colaterais.

Rotinas podem chamar outras sub-rotinas, ou a si mesmas recursivamente. Conforme elas são executadas elas alteram os valores das variáveis - estruturas de dados - que compõem o programa, de tal forma que a ordem de execução importa.



PARADIGMAS

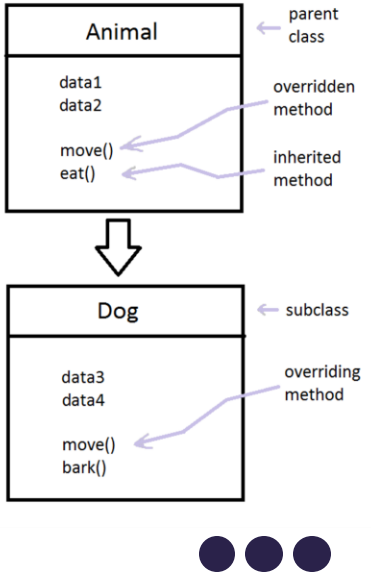
PROGRAMAÇÃO ORIENTADA À OBJETOS

Classe: É a **definição** de um formato de dados (um tipo personalizado) e procedimentos (blocos de código) disponíveis para este modelo de entidades e suas subclasses. – Uma subclasse pode expandir, alterar e especificar os atributos e comportamento de uma classe; chamamos essa técnica de extensão ou **herança**.

Objeto: Nada mais é que a **instância** de uma classe. Isto é, enquanto a classe especifica o esqueleto (as características) de um certo conjunto de variáveis, o objeto define os reais valores e ações de uma variável.

POO permite uma forma organizacional das entidades tirando proveito da abstração. A abstração é um dos conceitos fundamentais que este paradigma está fundado, além dos seguintes "pilares" a seguir:

- Herança: permite estabelecer relacionamentos hierárquicos entre classes. Isto é, uma classe pode derivar outra.
- Polimorfismo: quando uma classe herda de outra, ela pode alterar a sua forma – modificando suas características base e comportamento. Em outros termos, é como se variássemos as características do modelo.
- Encapsulamento: determina restrições sobre o acesso das informações (propriedades) entre classes e suas instâncias.



05. INTUIÇÃO MATEMÁTICA

Como a matemática se relaciona com jogos

INTUIÇÃO MATEMÁTICA

MATEMÁTICA & ABSTRAÇÃO

Ao longo da nossa apresentação, foi muito usado o termo **abstração**. Este termo é definido formalmente como um processo conceitual onde regras e conceitos são **derivados** do seu uso, classificações, exemplos específicos e princípios.
É geralmente visto como "algo difícil", assim como sua própria definição é complicada de se compreender a princípio. Mas é uma importante ferramenta do pensamento para **simplificação** de ideias concretas em conceitos generalizados.

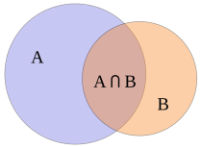
A matemática é uma das áreas do conhecimento que faz uso direto e intensivo da abstração, explorando o campo dos números (aritmética), fórmulas e estruturas (álgebra), formas e o espaço (geometria), quantidades e mudanças (análise).
É uma ferramenta usada em várias outras áreas, popularmente chamadas de ciências exatas, que nos permite compreender melhor as ideias que moldam o nosso universo.

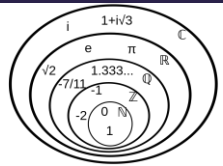
Na programação de jogos a matemática é indispensável, embora muitos programadores se dediquem aos conceitos de design e engenharia para atingir seus objetivos ela faz parte da fundação – e graças a abstração, o uso dos seus conceitos pode ser simplifica ocultando os detalhes de implementação.

INTUIÇÃO MATEMÁTICA

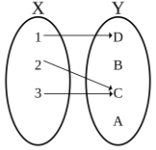
CONCEITOS FUNDAMENTAIS

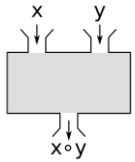
Conjunto: é um modelo matemático para uma coleção de diferentes coisas. Um conjunto contém elementos que o constituem. A teoria dos conjuntos é uma área da matemática que se dedica a entender os números e as coleções, e serve de base a matemática como um todo.

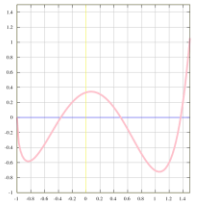




Funções: originalmente era idealizada como o estudo da variação de quantidades, mas em sua definição trata-se, em geral, da relação entre elementos de um dado conjunto X (domínio) para elementos de um outro conjunto Y (contradomínio).

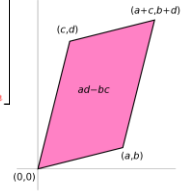




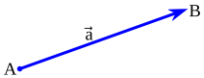


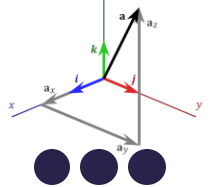
Matrizes: Uma matriz é um array retangular ou tabela com elementos arranjados em linhas e colunas, ela é usada para representar um objeto matemático ou propriedades sobre tais objetos.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \end{array}$$



Vetores: A álgebra vetorial e linear são os ramos da matemática que usam esse objeto matemático como ferramenta atuante. Vetores são muitas vezes representados por meio de matrizes.



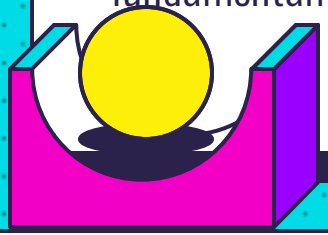


INTUIÇÃO MATEMÁTICA

GEOMETRIA E FÍSICA

Usando as ferramentas matemáticas e por meio de observações testadas com lógica argumentativa, os matemáticos conseguem estudar e descrever todas as **formas** no espaço precisamente.

- A geometria é o campo de estudos que investiga as características do espaço, desde a menor unidade (o ponto) até unidades extradimensionais.
- A física é uma ciência natural que investiga a natureza. Não se restringe apenas as dinâmicas do espaço-tempo, mas, em especial no campo da **mecânica** estudamos os mecanismos que fundamentam o universo, que é aquilo que estamos mais interessados na programação de – construir mundos virtuais com suas próprias naturezas particulares fundadas e inspiradas nas propriedades e restrições da realidade.



INTUIÇÃO MATEMÁTICA

MODELAGEM

Assim como é importante para um programador construir um grande know-how de conhecimentos de computação – estruturas de dados, design de algoritmos, arquitetura, engenharia, etc... – é também importante desenvolver a sua intuição matemática. Assim desenvolvendo suas capacidades de trabalho com a modelagem dos meios virtuais. Em especial, para programadores de jogos:

- Modelagem geométrica-física-espacial: desenvolvemos jogos que se passam em espaços virtuais, que são moldados de forma matemática e **visual**.
- Modelagem física-tempo-espaço (cinemática): não apenas restrito a jogos virtuais, o tempo é a medida que permite o ato de **jogar**, é o conceito fundamental da vida. Estudar a passagem do tempo permite os programadores darem vida a seus mundos imaginários.

INTUIÇÃO MATEMÁTICA

MATEMÁTICA E DESIGN

Como já foi dito, o objetivo de um programador é criar soluções para resolver problemas. Na programação de jogos, os problemas são criados pelo próprio programador – e é este fato que identifica software como arte.

A matemática é um canivete-suíço para resolução de problemas. Podemos listar alguns deles...

- **Problemas de contagem**: muitos jogos possuem diversas entidades que precisam ser mensuráveis para criar a possibilidade de ganhar ou perder.
- **Aleatoriedade e probabilidade**: jogos, em especiais procedurais, utilizam da aleatoriedade como ferramenta para simular a dinâmica caótica da natureza.
- **Estatísticas e gerenciamento de recursos**: programadores também pode estudar as estatísticas para melhor compreender como jogadores interagem com seus jogos, fazendo melhor uso dos recursos maximizando seu objetivo de jogabilidade (geralmente, maximizando a diversão).
- **Regras de jogo**: Podemos usar de ferramentas matemáticas para se criar regras em um dado contexto de jogo, e manter estas regras justas e claras para os jogadores neste estudo.
- **Design de puzzles**: Todos os outros conceitos matemáticos se aplicam aqui. Desenvolver jogos é divertido, pois resolver problemas é satisfatório, e os desenvolvedores sabem disso tal qual constroem seus próprios problemas para seus jogadores resolvê-los.

INTUIÇÃO MATEMÁTICA

GAME: BRAID

00. CRIATIVIDADE

O talento que qualquer um pode aprender

CRIATIVIDADE

USANDO FERRAMENTAS

A prática leva a perfeição.

Embora seja debatível se perfeição é algo alcançável, uma coisa é certa, para dominar algo você deve praticar exaustivamente. Existe um dizer popular de que para se tornar um mestre excepcional em algum talento você precisa 10.000 horas de dedicação. Na realidade, não existe uma medida específica de tempo para tal, alguns levam mais tempo e outros menos, mas é certo que dedicando-se de tal forma, você eventualmente irá "platinar" seu estudo.

Para fazer jogos, você não necessariamente precisa ser um programador, ou muito menos dominar a matemática, sequer ser um designer astuto... **Basta fazer jogos!!!**

Existem diversas ferramentas que ajudam a alcançar este objetivo. Dominando tais ferramentas, você pode criar e evoluir a sua criatividade. Não há restrições para o que é possível no mundo da imaginação, apenas aquelas que você se impõe.

USANDO FERRAMENTAS

UM NÚMERO ENORME DE OPÇÕES

Busque por “game dev tools” e diversas opções aparecerão para você

CRIATIVIDADE

APLICANDO CONCEITOS

A criatividade provém de:

- Ter um objetivo;
- Consumir bastante conteúdo;
- Associar esse conteúdo com seu objetivo;
- Imaginar e reinventar
- Praticar bastante!

O conhecimento nos permite compreender melhor o mundo, e entendendo o que está ao nosso redor podemos ter referências mais ricas em detalhes que fazem a diferença na construção de artefatos relevantes e experiências memoráveis.

Ser um artista não se resume a conceitos, mas o que você faz com as ideias.

CRIATIVIDADE

DESIGN DE JOGOS

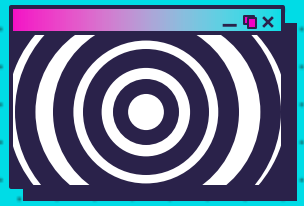
O campo de design de jogos é rico de estudos, técnicas, ferramentas, discussões e modelos que nos permitem absorver e aplicar as ideias de forma construtiva e experimental.

Projetar jogos é uma experiência divertida e desafiadora. Trata-se de formular experiências lúdicas interessantes que são construídas na imaginação de outros.

Criar jogos é um ato significativo de paixão.

— EVERY INDIE DEVELOPER EVER

ENCERROU!



PERGUNTAS?

Email de contato: gerson.creative.des@gmail.com
Agradecimentos: Miguel Reis, Daniel Nascimento

FINISHED!

PRÓXIMAS SEÇÕES

TEORIA

PRÁTICA

Desenvolvendo o jogo Flappy Bird

INTRO

GDSCRIPT

Introdução breve à linguagem da Godot