



departamento de
**EXPRESSÃO
GRÁFICA**

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE CIÊNCIAS EXATAS
DEPARTAMENTO DE EXPRESSÃO GRÁFICA
Professor: Paulo Henrique Siqueira
Disciplina: Tópicos em Visualização Científica



Atribuição-Não Comercial-Sem Derivações: CC BY-NC-ND

VISUALIZAÇÃO CIENTÍFICA

1. Introdução

Quando pensamos na palavra **Visualizar**, podemos encontrar na literatura as seguintes definições:

- tornar visível;
- tornar visual;
- ver uma imagem mentalizada;
- figurar mentalmente;
- converter algo abstrato em imagem mental ou real; ou
- tornar algo visível mediante algum recurso.

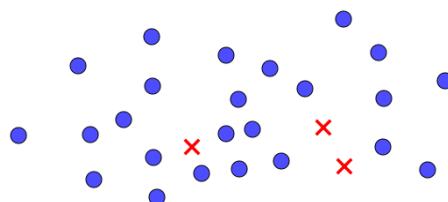
Se procurarmos um significado para a palavra **Visualização**, encontramos as seguintes definições:

- capacidade de formar na mente imagens visuais de coisas que não estão visíveis;
- transformação de conceitos abstratos em imagens reais ou mentalmente visíveis;
- conversão de números ou dados para o formato gráfico; ou
- visualidade (qualidade ou estado de ser visual ou visível).

De acordo com Card, Mackinlaye Shneiderman (1999) e Eler (2020), a visualização não está relacionada ao computador: a finalidade da visualização é a compreensão do indivíduo, não as imagens. Os principais objetivos desta compreensão são:

- realizar descobertas;
- verificar hipóteses;
- auxiliar a tomada de decisões; e
- explicar questões concretas.

Podemos dizer que a **Visualização** está relacionada com a cognição do ser humano. Visualizar é algo que fazemos naturalmente e o sistema visual humano é o sentido com maior capacidade de captação de informações por unidade de tempo, que tem processamento rápido e paralelo e está treinado para reconhecer padrões. No exemplo mostrado a seguir, quantos padrões "X" estão representados no conjunto?

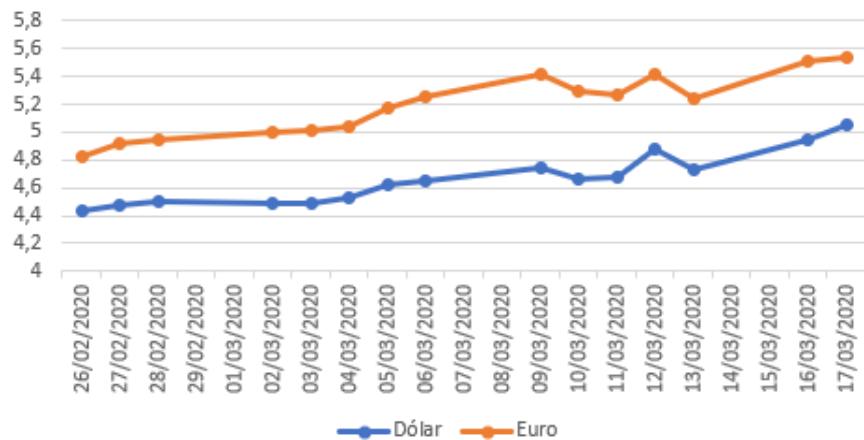


Quando pesquisamos cotações de moedas estrangeiras, podemos visualizar estes dados de algumas maneiras. A mais comum é de uma tabela, onde podemos restringir o intervalo de tempo. O exemplo mostrado a seguir apresenta as cotações do dólar e do euro no período de 15 dias entre os dias 26/02 e 17/03/2020.

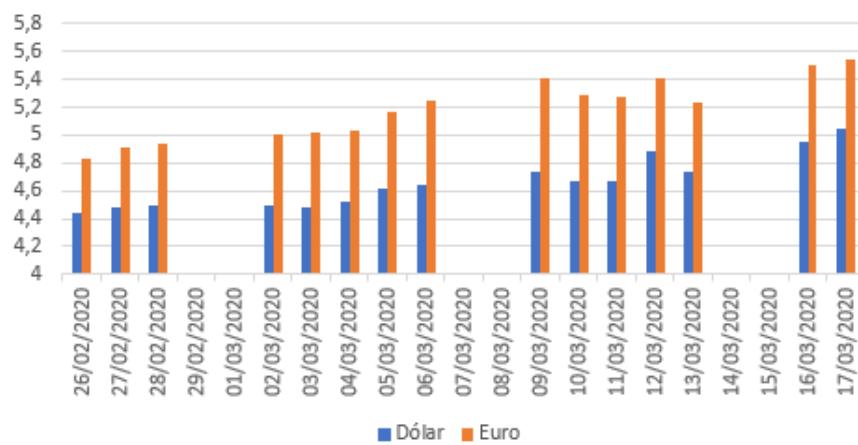
Data	26/02/2020	27/02/2020	28/02/2020	02/03/2020	03/03/2020	04/03/2020	05/03/2020	06/03/2020	09/03/2020	10/03/2020	11/03/2020	12/03/2020	13/03/2020	16/03/2020	17/03/2020
Dólar	4,4353	4,4758	4,4981	4,494	4,4877	4,5252	4,6201	4,6453	4,7373	4,6687	4,6732	4,6825	4,7355	4,9464	5,0489
Euro	4,8247	4,9158	4,9403	5,0014	5,0159	5,0393	5,1676	5,2534	5,4109	5,2948	5,2704	5,4098	5,2413	5,5078	5,5391

Disponível em: <https://www.bcb.gov.br/estabilidadefinanceira/historicocotacoes>

Podemos visualizar estes dados com gráficos lineares, que são bem comuns em várias páginas da internet que mostram dados financeiros.



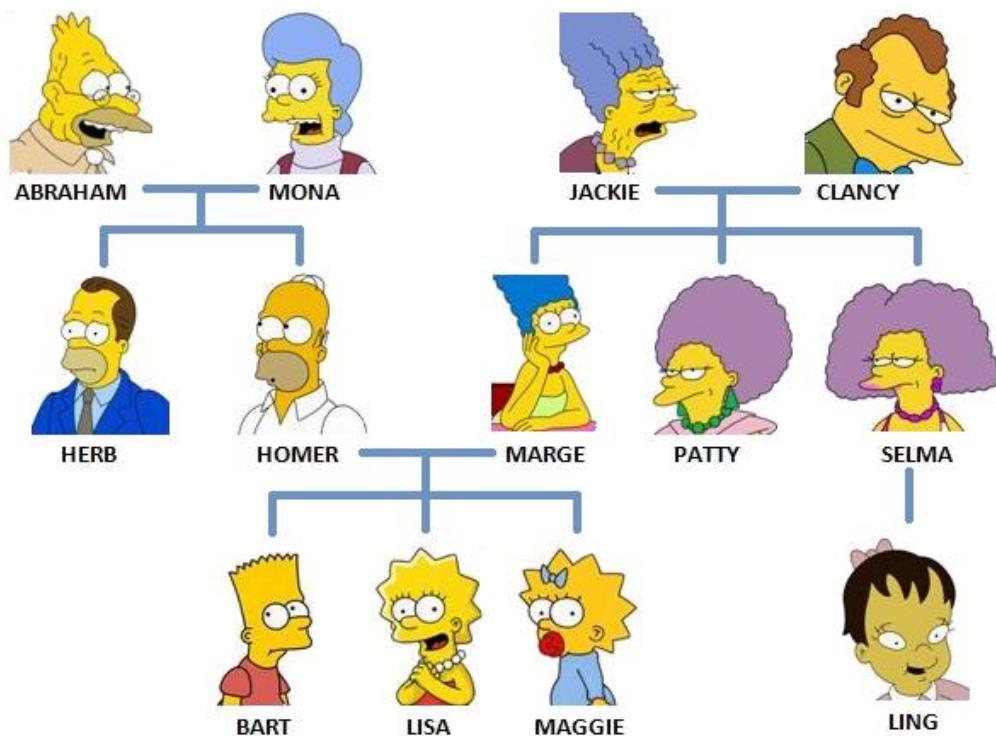
Podemos visualizar estes dados de outra forma, utilizando os gráficos de barras.



No próximo exemplo, temos os dados da árvore genealógica da família dos Simpsons colocados em uma tabela.

Pessoa	Mãe	Pai
Herb	Mona	Abraham
Homer	Mona	Abraham
Marge	Jackie	Clancy
Patty	Jackie	Clancy
Selma	Jackie	Clancy
Bart	Marge	Homer
Lisa	Marge	Homer
Maggie	Marge	Homer
Ling	Selma	

Organizando estes dados como um organograma, respeitando as ascendências e descendências mostradas na tabela, podemos visualizar a árvore genealógica de uma forma mais adequada.



De acordo com os dados do IBGE de 2019, podemos relacionar o índice de alfabetização de cada estado com sua respectiva renda per capita. Podemos analisar a tabela com estes dados, detectando estas relações e a existência de outliers, que são exceções ou discrepâncias.

Comparativo da renda per capita e do índice de alfabetização de cada estado brasileiro em 2019

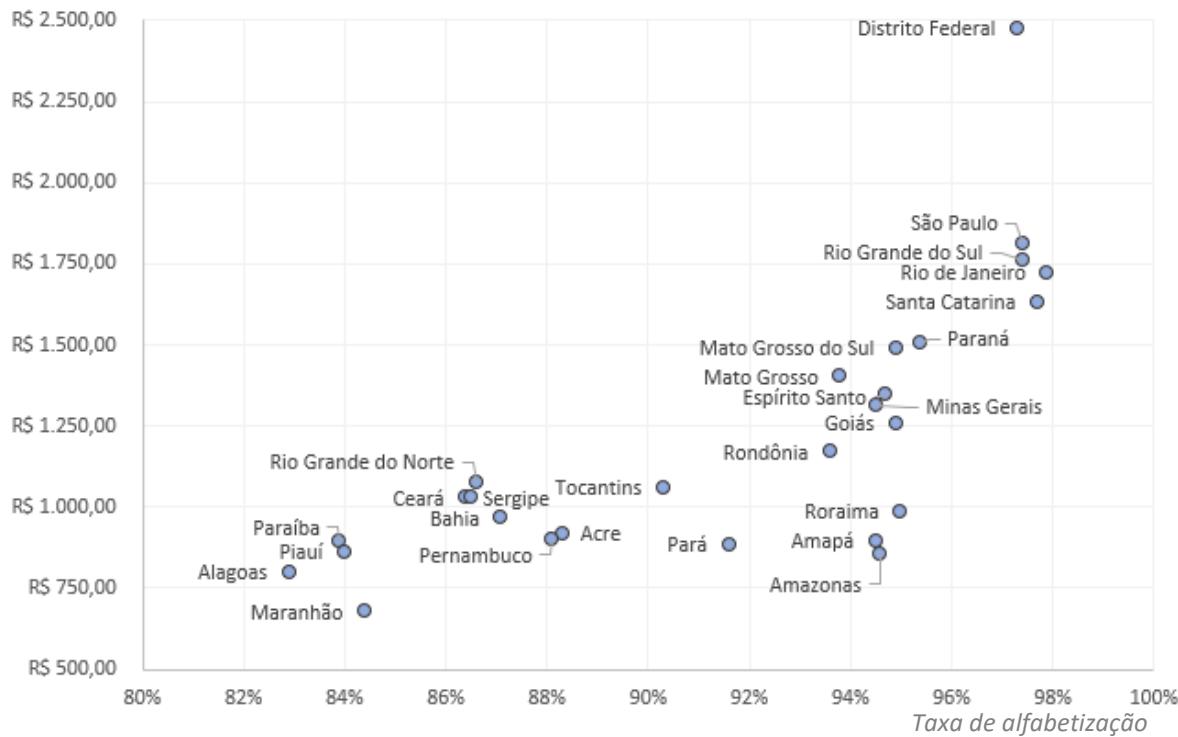
Estado	Renda per capita	Índice de alfabetização	Estado	Renda per capita	Índice de alfabetização
Acre	R\$ 917,00	88,3%	Paraíba	R\$ 892,00	83,9%
Alagoas	R\$ 796,00	82,9%	Paraná	R\$ 1.508,00	95,4%
Amapá	R\$ 893,00	94,5%	Pernambuco	R\$ 897,00	88,1%
Amazonas	R\$ 852,00	94,6%	Piauí	R\$ 859,00	84,0%
Bahia	R\$ 965,00	87,1%	Rio de Janeiro	R\$ 1.723,00	97,9%
Ceará	R\$ 1.028,00	86,4%	Rio Grande do Norte	R\$ 1.077,00	86,6%
Distrito Federal	R\$ 2.475,00	97,3%	Rio Grande do Sul	R\$ 1.759,00	97,4%
Espírito Santo	R\$ 1.347,00	94,7%	Rondônia	R\$ 1.169,00	93,6%
Goiás	R\$ 1.258,00	94,9%	Roraima	R\$ 983,00	95,0%
Maranhão	R\$ 676,00	84,4%	Santa Catarina	R\$ 1.632,00	97,7%
Mato Grosso	R\$ 1.401,00	93,8%	São Paulo	R\$ 1.814,00	97,4%
Mato Grosso do Sul	R\$ 1.488,00	94,9%	Sergipe	R\$ 1.028,00	86,5%
Minas Gerais	R\$ 1.314,00	94,5%	Tocantins	R\$ 1.060,00	90,3%
Pará	R\$ 883,00	91,6%			

Disponível em: <https://www.ibge.gov.br>

Se transformarmos os dados desta tabela em um gráfico, usando as informações das taxas de alfabetização como eixo x e as informações de renda per capita de cada estado como eixo y, podemos analisar estes dados de uma maneira mais rápida e eficiente.

Comparativo da renda per capita e do índice de alfabetização de cada estado brasileiro em 2019

Renda per capita



Disponível em: <https://www.ibge.gov.br>

Agora podemos definir o que é a **visualização**: trata-se da representação de um conjunto por meio de gráficos, imagens, animações e interações para apresentar informações ou comportamento de dados, estruturas ou maquetes. As técnicas de visualização são usadas para compreender os dados e extraer conhecimento (TELEA, 2015; WILLIAMS, SOCHATS e MORSE, 1995; ELER, 2020).

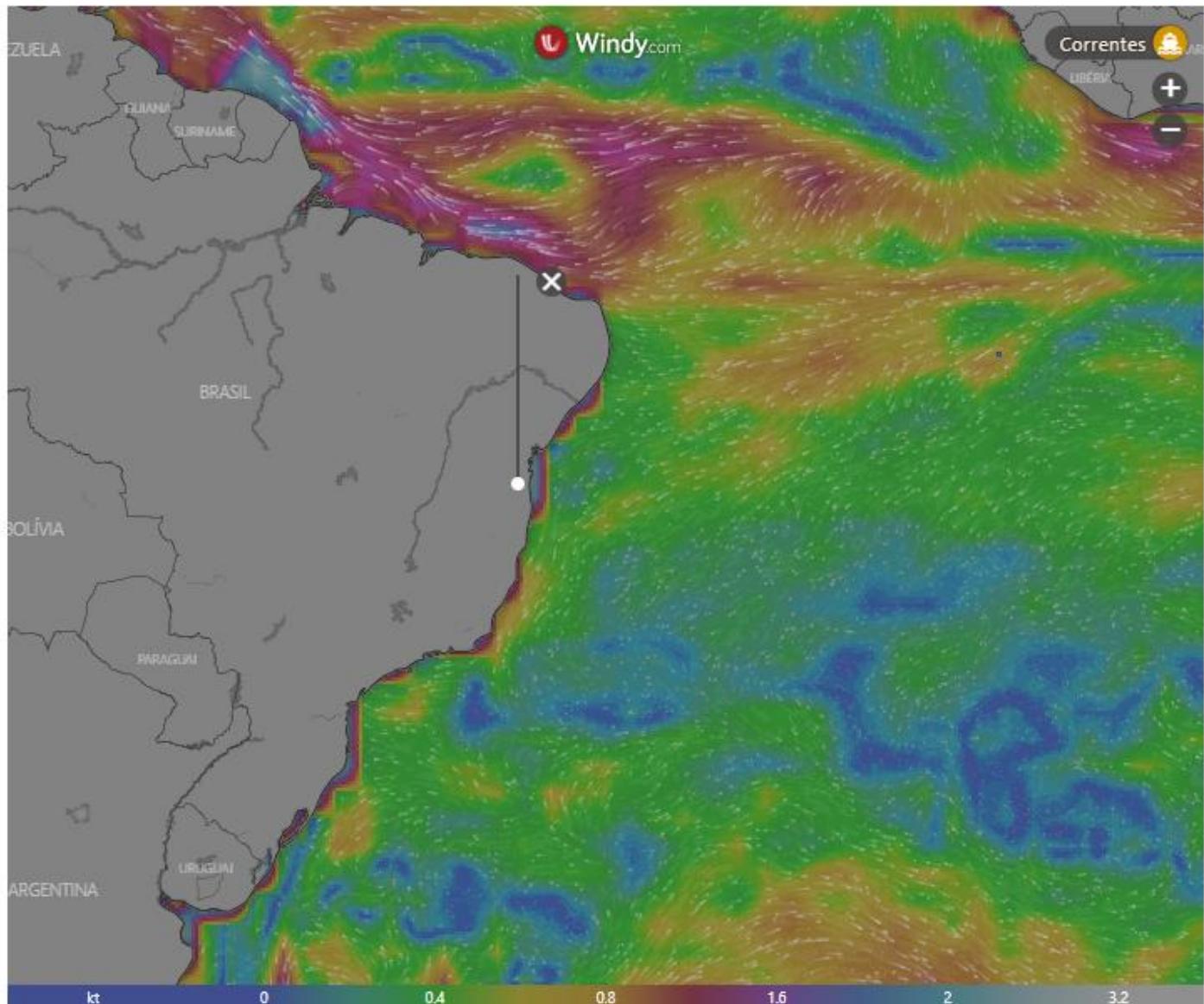
De acordo com os exemplos apresentados, podemos concluir que a **visualização** é a comunicação da informação que utiliza representações gráficas. Nestas representações, as imagens podem ser usadas como mecanismos de comunicação e uma única imagem pode conter várias informações que podem ser processadas mais rapidamente do que uma página ou tabela com palavras e números (WARD, GRINSTEIN e KEIM, 2010).

Em algumas situações fica praticamente impossível olhar uma tabela e verificar todas as informações para tomar uma decisão ou comparar os dados. Considere o exemplo da representação do fluxo de calor no Brasil. Para cada ponto do mapa, temos as informações das coordenadas geográficas e a temperatura aproximada deste ponto. Porém, somente com a visualização gráfica do mapa, com as temperaturas representadas com tons diferentes de cores que conseguimos ter a visão geral do fluxo de temperatura.

Em uma situação um pouco mais complexa, podemos imaginar quantas informações de uma tabela devem ser usadas para a construção de um mapa com as correntes marítimas na costa brasileira. Assim como em mapas de fluxo de calor, as correntes são representadas com diferentes tons de cores que indicam suas respectivas velocidades (geralmente com a unidade nó – kt). Além disso, temos as indicações do sentido de cada corrente que são feitos por meio de setas sobre as representações gráficas das correntes marítimas.

Veja o exemplo mostrado a seguir que está na página Climatologia Geográfica. Nesta tela, podemos escolher outras representações de gráficos que envolvem temperatura, ventos, tempestades e outras informações importantes que seriam praticamente impossíveis de visualizarmos apenas com tabelas. Estes mapas contêm as informações em tempo real.

Correntes marítimas brasileiras



Disponível em: <https://climatologiageografica.com/previsao-atmosferica-em-tempo-real/>

Os objetivos da visualização científica de dados são os seguintes:

- Armazenar informações;
- Analisar dados para apoiar e ajudar o raciocínio;
- Confirmar hipóteses;
- Mostrar as ideias para outras pessoas; e
- Tomada de decisões.

Resumindo, a visualização tem o objetivo de proporcionar uma espécie de *insight* utilizando representações gráficas interativas que consideram vários aspectos relacionados a algum processo ou fenômeno observado.

As fontes de dados nunca se esgotam. A cada minuto temos bilhões de dados gerados no mundo digital que mostram como nos relacionamos uns com os outros e até mesmo como nos relacionamos com as marcas e o mundo digital. Os dados produzidos digitalmente não mostram sinais de desaceleração.

Veja o que aconteceu em cada minuto no mundo digital do ano de 2021, mostrado com os dados do infográfico mostrado a seguir. Estas informações foram coletadas para a 9ª Edição de **Data Never Sleeps**.

Infográfico Data Never Sleeps do ano 2021



Disponível em: <https://www.domo.com/learn/infographic/data-never-sleeps-9>

As motivações para o estudo da Visualização Científica são as seguintes:

- facilidade de coletar e armazenar dados;
- dificuldade de processar, analisar e interpretar todos os dados coletados;
- identificar e estabelecer os dados que são relevantes;
- fenômenos que possuem volumes muito grandes de dados;
- dimensionalidade muito grande dos dados;
- natureza diversificada dos dados (registros de textos, imagens, vídeos ou áudios); e
- desafio dos pesquisadores de Visualização Científica no auxílio da análise de dados.

A área da Visualização tem importância fundamental nas pesquisas científicas, pois podemos trabalhar com estatísticas, mineração de dados, aprendizado de máquina e outras técnicas. A Visualização possibilita a exploração de questões que não são diretamente efetuadas com base somente nos dados. Além disso, a Visualização auxilia na formulação de novas questões relativas aos dados. Por exemplo, distribuições, correlações, tendências e as técnicas de Pesquisa Operacional e Inteligência Artificial são mais bem compreendidas quando visualizadas.

Um exemplo que podemos observar é do Quarteto de Anscombe (1973), que consiste em 4 conjuntos de dados que têm estatísticas descritivas praticamente iguais, mas com distribuições e aparências distintas quando exibidas graficamente.

Os quatro conjuntos de dados de Anscombe

1		2		3		4	
x	y ₁	x	y ₂	x	y ₃	x	y ₄
10	8,04	10	9,14	10	7,46	8	6,58
8	6,95	8	8,14	8	6,77	8	5,76
13	7,58	13	8,74	13	12,74	8	7,71
9	8,81	9	8,77	9	7,11	8	8,84
11	8,33	11	9,26	11	7,81	8	8,47
14	9,96	14	8,1	14	8,84	8	7,04
6	7,24	6	6,13	6	6,08	8	5,25
4	4,26	4	3,1	4	5,39	19	12,5
12	10,84	12	9,13	12	8,15	8	5,56
7	4,82	7	7,26	7	6,42	8	7,91
5	5,68	5	4,74	5	5,73	8	6,89

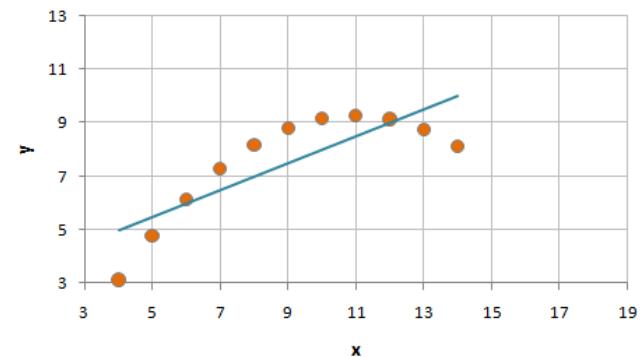
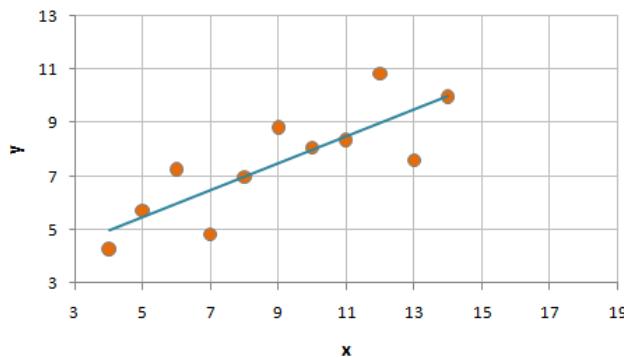
Analisando as Estatísticas Descritivas dos 4 conjuntos de dados de Anscombe, temos os seguintes valores:

Estatísticas descritivas dos dados de Anscombe

Propriedade	Valor	Precisão
Média de x	9	exato
Variância de x	11	exato
Média de y	7,50	até 2 casas decimais
Variância de y	4,125	±0,003
Correlação entre x e y	0,816	até 3 casas decimais
Reta de Regressão Linear	$y = 0,5x + 3,0$	até 3 casas decimais
Coeficiente de determinação da regressão linear: R ²	0,67	até 2 casas decimais

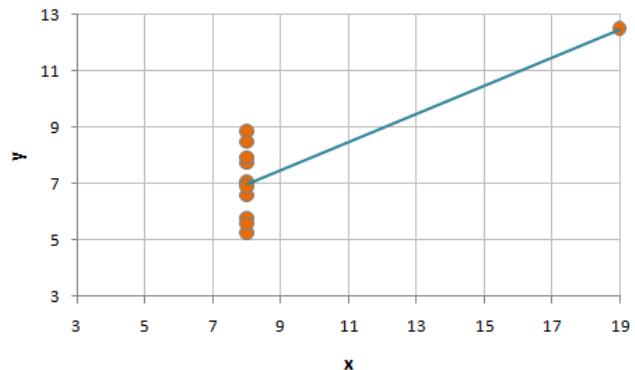
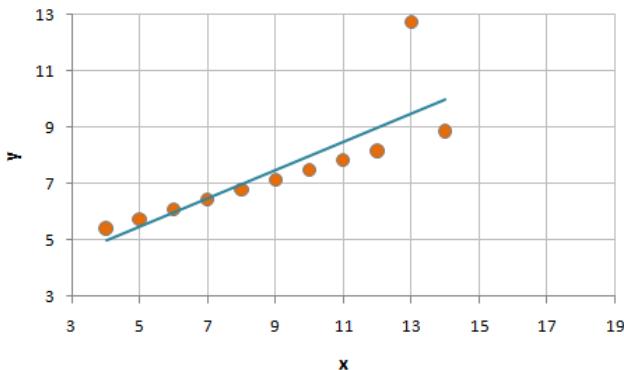
Porém, ao analisar graficamente estes conjuntos de dados, podemos verificar a importância da Visualização Científica para estes conjuntos de dados.

Visualização dos dados dos Conjuntos 1 e 2 de Anscombe



Uma análise superficial, levando-se em conta apenas as Estatísticas Descritivas, poderia levar ao leitor à conclusão precipitada de que são conjuntos idênticos (ou muito similares).

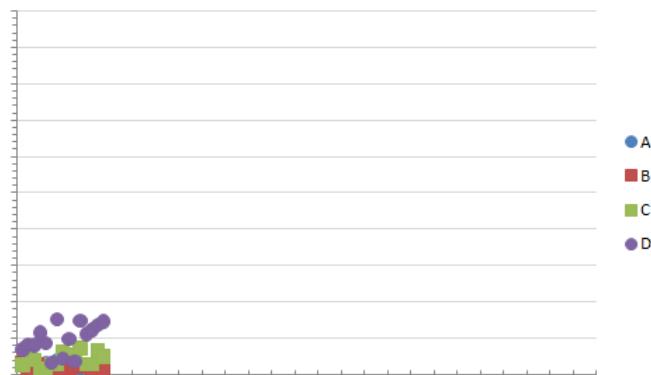
Visualização dos dados dos Conjuntos 3 e 4 de Anscombe



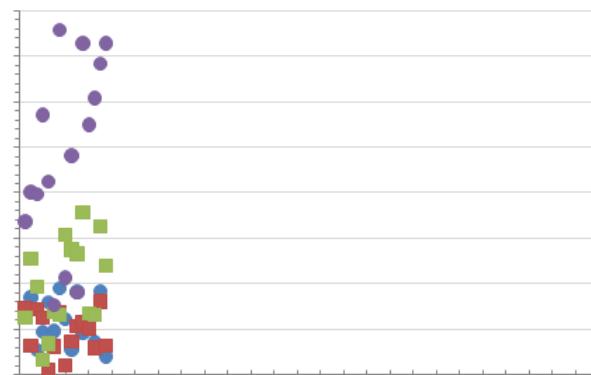
Encontramos a Visualização de dados para ajudar ou substituir conjuntos de informações no nosso dia-a-dia: tabelas em sites de notícias, mapas de linhas de transporte público, mapas climáticos, diagnósticos médicos, dados financeiros ou pesquisas de opinião.

A visualização apresenta uma alternativa para a informação textual ou verbal, ou seja, a representação gráfica acaba proporcionando descrições mais detalhadas das informações. Nestes casos, o processamento da informação é feito de maneira paralela, ao invés da sequencial com textos, vídeos ou áudios.

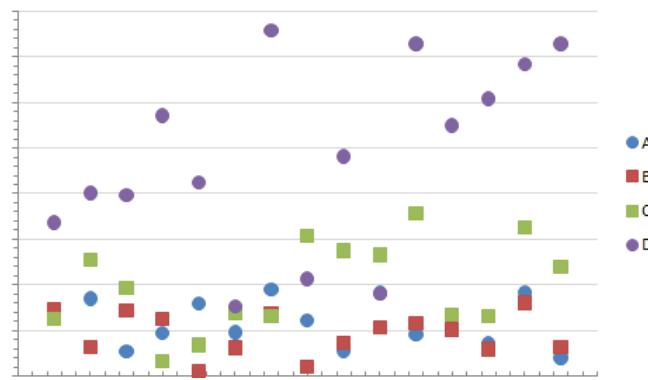
Porém, não basta apenas construir gráficos ou sistemas interativos visuais; precisamos estudar a melhor maneira para apresentar dados para os leitores. Note o seguinte exemplo, com dados plotados em uma escala.



Os leitores não conseguem diferenciar os 4 conjuntos de dados com esta visualização. Ajustando-se a escala do eixo y, conseguimos uma visualização melhorada destes conjuntos de dados.

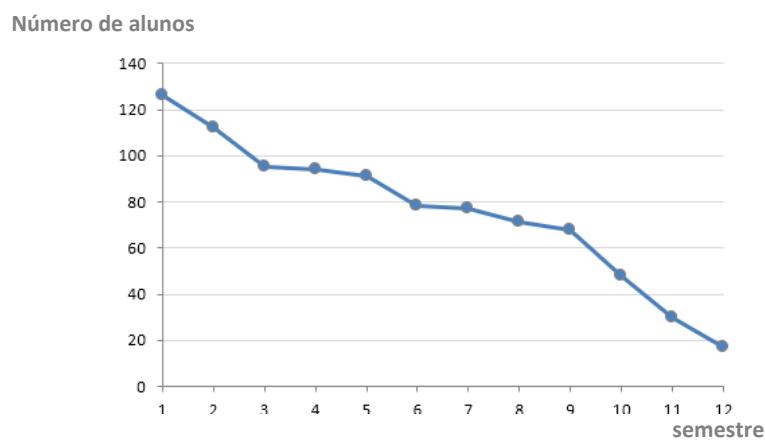


Porém, muitos dados ficam agrupados em uma faixa próxima ao eixo y. Somente com o ajuste no eixo x temos uma visualização mais adequada destes conjuntos de dados.

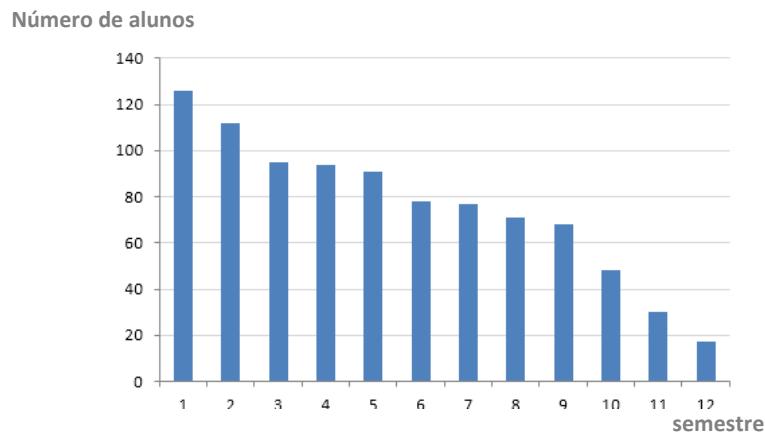


Outros estudos envolvem a definição da melhor maneira de apresentar os dados graficamente. O exemplo mostrado a seguir contém os mesmos dados representados com três maneiras distintas de gráficos com duas dimensões.

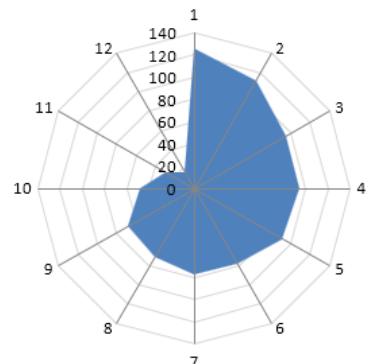
Número de alunos matriculados em um curso de graduação por semestre – dispersão em X e Y com linhas



Número de alunos matriculados em um curso de graduação por semestre – gráfico em colunas



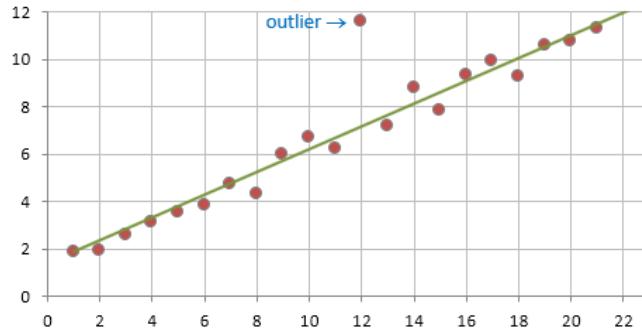
Número de alunos matriculados em um curso de graduação por semestre – gráfico radar



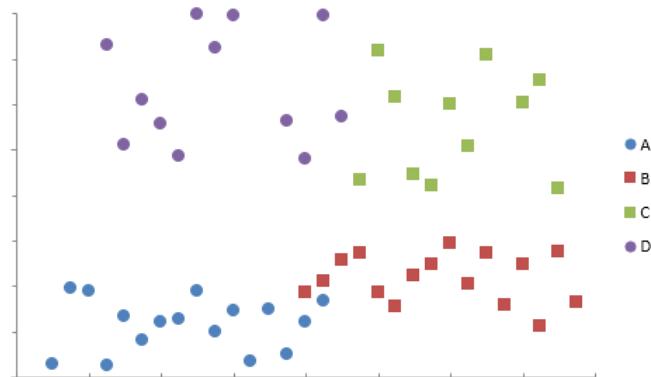
Quando vamos construir uma forma de visualização, devemos analisar o tipo de dados disponíveis e qual seria o tipo de informação que o usuário espera extrair deste conjunto de dados. Devemos lembrar que o usuário utiliza a visualização para:

- exploração – procura de alguma informação ou comportamento interessante dos dados;
- confirmação de hipóteses – valida resultados de análises quantitativas ou limites dos dados; e
- apresentação de resultados – analisar os dados e mostrar informações para um público específico.

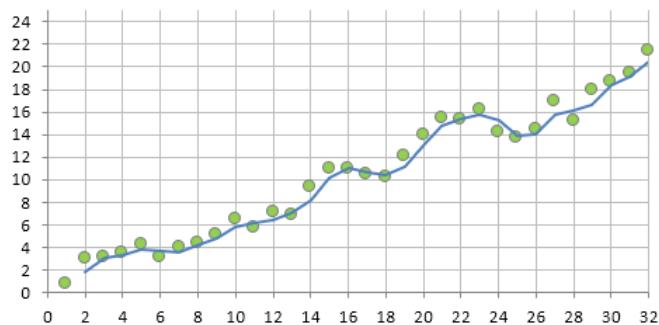
Na visualização de um conjunto podemos determinar erros ou anomalias (outliers). Por meio dos dados do gráfico, podemos detectar erros ou determinar características especiais destes dados que ficam “fora da curva”.



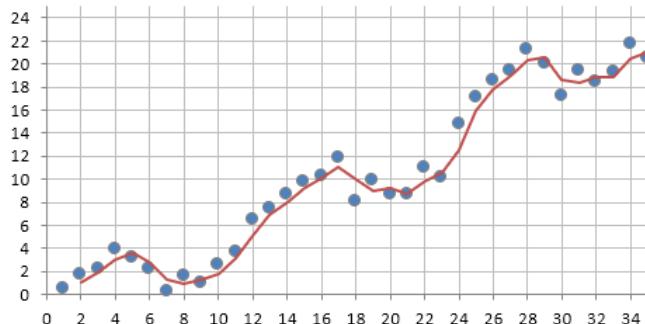
De acordo com as posições representadas de subgrupos de um conjunto, podemos observar a criação de agrupamentos de dados (dados com características similares). Se utilizarmos somente as observações destes dados em tabelas, levaremos mais tempo para detectar as formações destes agrupamentos de dados.



Podemos observar também mudanças de tendências nos gráficos de conjuntos de dados.



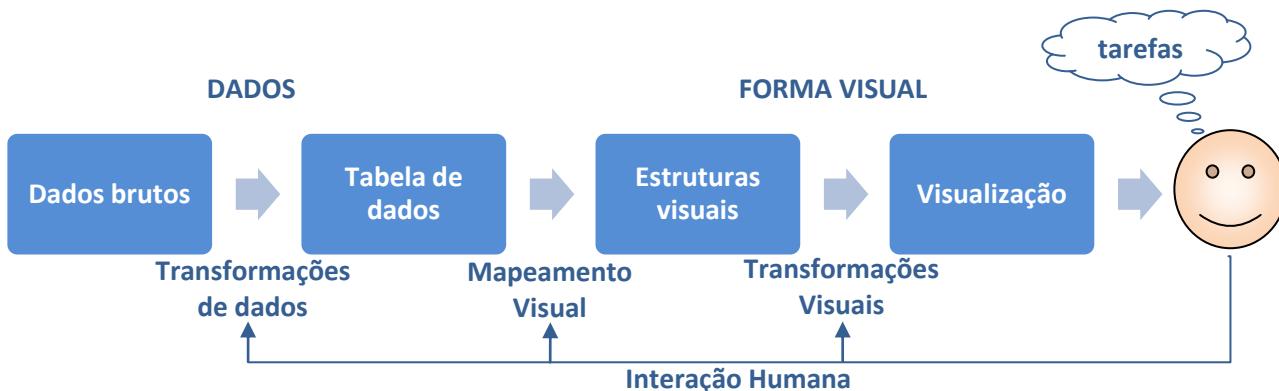
Em conjuntos de dados que envolvem observações temporais, podemos observar efeitos de sazonalidade dos dados representados graficamente.



1.1. Etapas do processo de Visualização

Para criarmos a visualização de um conjunto de dados é necessário definirmos como este conjunto será mapeado para representá-lo graficamente. Vamos compreender as etapas do processo de visualização utilizando um **pipeline**. Este procedimento consiste de uma sequência de etapas de estudos independentes em relação aos algoritmos e estruturas.

Podemos resumir o **pipeline** que define as etapas da visualização por meio da representação em fluxograma apresentada a seguir:



A primeira etapa é a de **transformações de dados**, que consiste em dois momentos.

Começamos com a **modelagem dos dados**, que consiste na **obtenção, preparação e organização** dos dados que devem ser visualizados. Esta fase é executada em arquivos, com os dados estruturados em um banco ou repositório de dados.

Depois de modelarmos os dados, devemos executar o **tratamento dos dados** por meio da **seleção** e da **identificação** de subconjuntos de dados que podem ser usados no processo de exploração dos dados.

Na etapa chamada de **mapeamento visual** podemos **identificar valores limites** dos dados para a **construção** das entidades gráficas e de seus atributos: cores, posições, formatos e tamanhos.

A etapa de **transformações visuais** consiste em dois momentos.

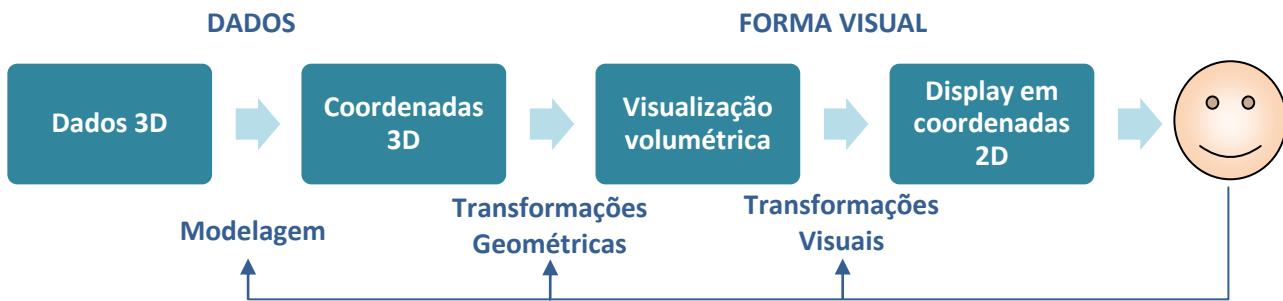
Começamos com as **configurações de parâmetros** da representação gráfica gerada, definindo a escala de cores, as dimensões (unidimensional, bidimensional ou tridimensional), formatos de gráficos, iluminação e os tipos de interação com os usuários.

Para finalizar esta etapa, temos a **renderização** ou geração da visualização. Neste momento, usamos as técnicas de criação da representação visual que deve ser explorada e analisada pelo usuário. Podemos definir efeitos de tonalização, mapeamentos de texturas, desenhos dos eixos, além de observações e dicas para o usuário.

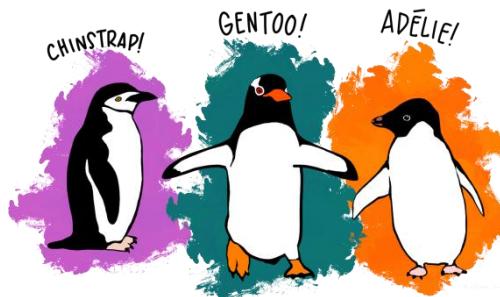
O objetivo do pipeline da **Computação Gráfica** é a síntese de imagens feita por meio das seguintes etapas:

- **Modelagem** dos objetos que serão representados;
- **Transformações Geométricas** na representação dos objetos; e

- **Transformações Visuais** da representação dos objetos em 3D para funcionar em um display 2D: projeção para fazer a renderização.



Vamos analisar um caso de visualização dos dados para classificação das espécies de pinguins. As espécies de pinguins são Chinstrap, Gentoo e Adélie (Horst, Hill e Gorman, 2020).

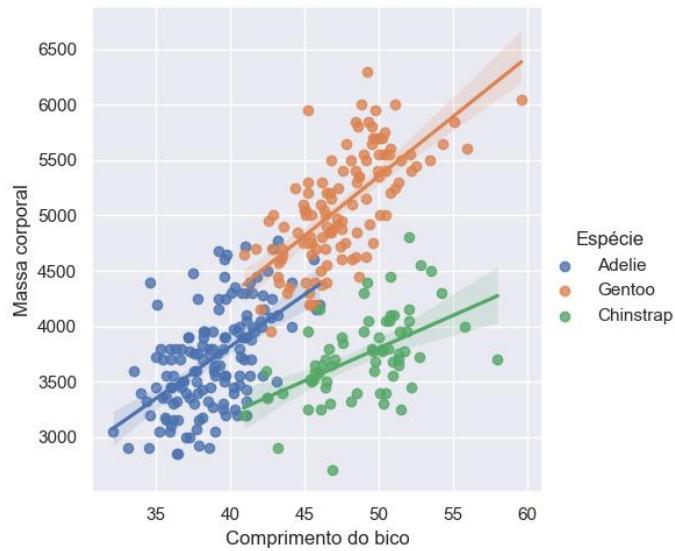


Fonte: https://inria.github.io/scikit-learn-mooc/python_scripts/trees_dataset.html

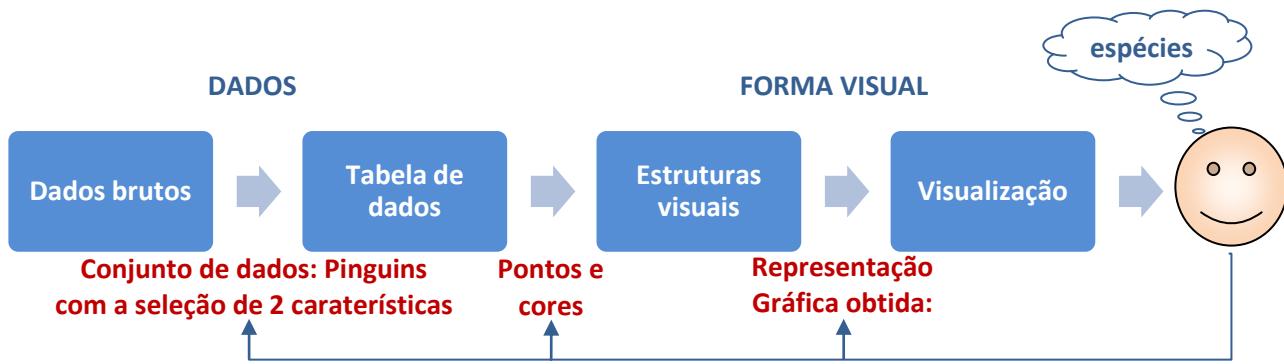
Podemos classificar os dados com base em combinações de características coletadas de cada espécie. O banco de dados possui dados de 344 pinguins coletados entre os anos 2007 e 2009.

<i>Id</i>	<i>Espécie</i>	<i>Ilha</i>	<i>Comprimento do bico</i>	<i>Profundidade do bico</i>	<i>Comprimento da nadadeira</i>	<i>Massa corporal</i>	<i>Sexo</i>	<i>Ano</i>
1	Adelie	Torgersen	39.1	18.7	181	3750	male	2007
2	Adelie	Torgersen	39.5	17.4	186	3800	female	2007
3	Adelie	Torgersen	40.3	18	195	3250	female	2007
4	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
5	Adelie	Torgersen	36.7	19.3	193	3450	female	2007
6	Adelie	Torgersen	39.3	20.6	190	3650	male	2007
7	Adelie	Torgersen	38.9	17.8	181	3625	female	2007
8	Adelie	Torgersen	39.2	19.6	195	4675	male	2007
9	Adelie	Torgersen	34.1	18.1	193	3475	NA	2007
10	Adelie	Torgersen	42	20.2	190	4250	NA	2007
...								
153	Gentoo	Biscoe	46.1	13.2	211	4500	female	2007
154	Gentoo	Biscoe	50	16.3	230	5700	male	2007
155	Gentoo	Biscoe	48.7	14.1	210	4450	female	2007
...								
342	Chinstrap	Dream	49.6	18.2	193	3775	male	2009
343	Chinstrap	Dream	50.8	19	210	4100	male	2009
344	Chinstrap	Dream	50.2	18.7	198	3775	female	2009

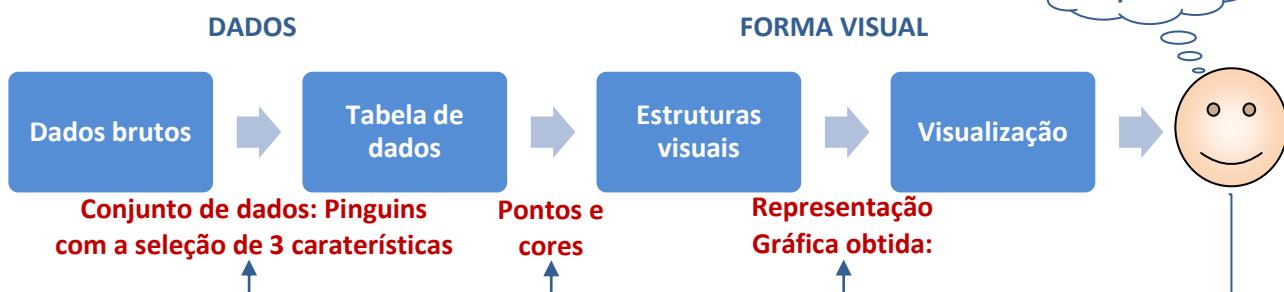
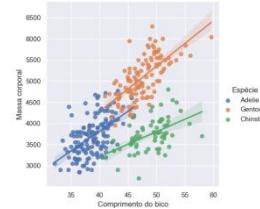
Ao escolher as características do Comprimento do bico e da massa corporal de cada espécie, podemos analisar a classificação em um gráfico 2D com as retas da regressão linear para cada espécie.



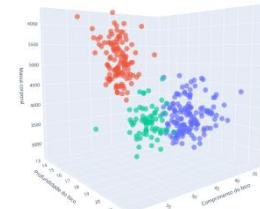
Desta forma, nosso pipeline da Visualização pode ser representado da seguinte maneira:



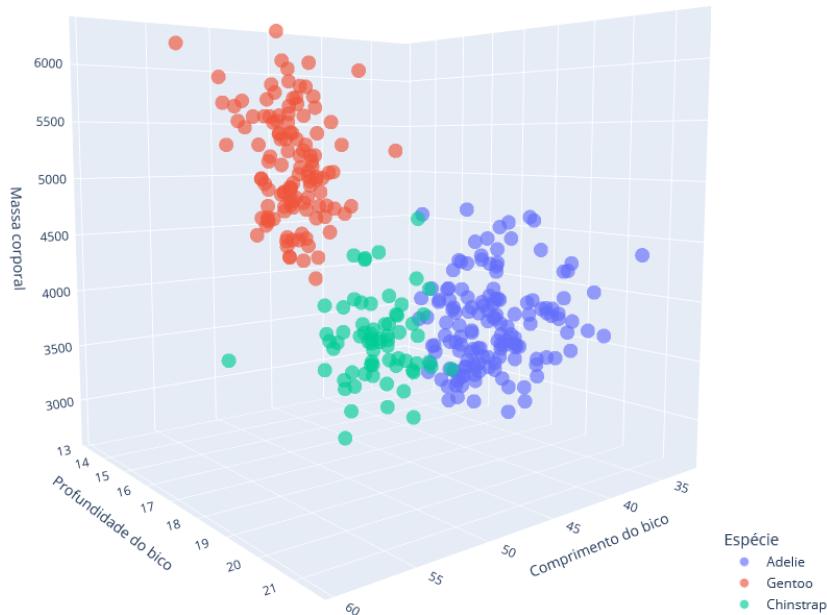
Nos casos em que estas duas características são insuficientes, podemos escolher mais uma característica para verificar a classificação por meio de um gráfico 3D?



Se escolhermos mais do que 3 características, devemos utilizar os critérios de redução de dimensionalidade para representar os dados em gráficos 2D ou 3D.



O gráfico 3D com as características Massa Corporal, Profundidade e Comprimento do bico de cada pinguim está representado a seguir.



Podemos concluir que a Visualização Científica utiliza os elementos e os recursos da Computação Gráfica para gerar as representações visuais dos dados. Logo, a Visualização Científica faz uma espécie de conexão entre uma representação gráfica e o respectivo conjunto de dados.

O foco da Computação Gráfica é a síntese de imagens, animação e entretenimento (exemplos: jogos, vídeos e filmes). No caso da Visualização Científica, temos uma ênfase na comunicação de informações presentes em conjuntos de dados por meio das tecnologias disponíveis da Computação Gráfica. A Visualização Científica envolve as áreas de Estatística, Mineração de dados, Inteligência Artificial, Percepção e Interação entre homem e máquina.

Atividade 1

Selecione 4 conjuntos de dados de classificação de padrões para utilizarmos nas próximas atividades da disciplina. Como utilizaremos os bancos de dados Iris e Pinguins em vários exemplos, selecione conjuntos diferentes destes, e que contenham pelo menos 3 variáveis usadas como critérios de classificação.

2. Conceitos básicos e estruturais de visualização

Nesta seção vamos apresentar os conceitos básicos que envolvem os elementos da Computação Gráfica que são necessários para utilizarmos nas bibliotecas de Visualização Científica.

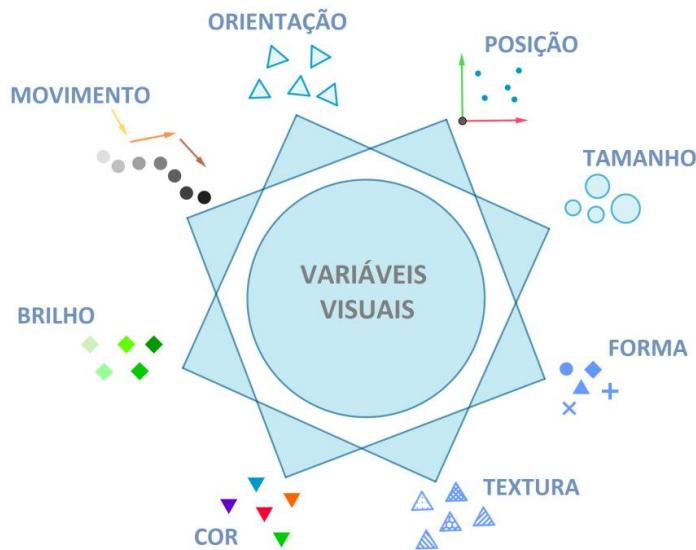
Os elementos mais simples da Computação Gráfica são os pontos. São os elementos sem dimensão, também chamados de 0-dimensionais. Um ponto A pode ser representado em gráficos de 2 dimensões A(x, y) ou com 3 dimensões A(x, y, z). As coordenadas de um ponto representam sua posição no plano ou no espaço.

Uma das formas que utilizamos para codificar dados em uma representação gráfica consiste no mapeamento de diferentes valores de dados para diferentes marcadores gráficos, modificando os seus atributos. Especificando os marcadores, podemos aplicar propriedades gráficas para cada um deles.

Um símbolo gráfico é definido como um objeto visual, que deve ser facilmente reconhecido e precisa refletir o que é apresentado no conjunto de dados. A ordem entre os dados pode ser representada por meio da ordem entre os símbolos. As similaridades entre os dados também devem representar as similaridades entre os símbolos. A interpretação é facilitada quando usamos elementos visuais que refletem diretamente as relações entre os dados.

Podemos considerar oito variáveis visuais para codificar uma informação: **Posição, Forma, Cor, Tamanho, Brilho, Orientação, Textura e Movimento**.

A **Posição** é uma das variáveis visuais mais importantes, pois o posicionamento de cada representação gráfica dos dados causa um grande impacto. Trata-se de um dos primeiros passos na leitura de uma visualização. Como o espaço da tela é limitado, ele deve ser bem aproveitado. A compreensão da distribuição de dados na tela é importante, pois podemos descobrir onde existe a concentração da maioria dos dados em uma região, ou se os dados possuem alguma distribuição Estatística, ou se existe alguma tendência ou estrutura visível nos dados.

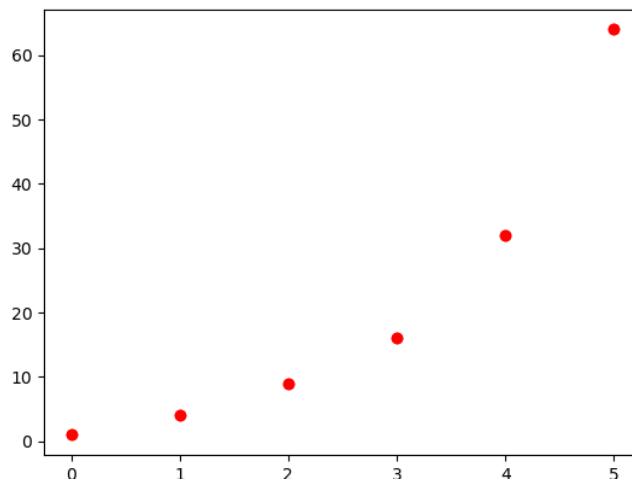


Considere o seguinte código em **C++** para representar 6 pontos por meio da biblioteca **matplotlib**.

```
#include <vector>
#include "matplotlibcpp.h"
Namespaceplt plt=matplotlibcpp;

int main(){
std::vector<double>x={0, 1, 2, 3, 4, 5};
std::vector<double>y={1, 4, 9, 16, 32, 64};
plt::scatter(x,y,{{"color","red"}, {"marker":"o"}});
plt::show();

return 0;
}
```



O código em **Python** para esta mesma representação gráfica é o seguinte:

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 32, 64]

plt.scatter(x, y, color = 'red', marker = 'o')
plt.show()
```

Os vetores x e y devem possuir a mesma dimensão e representam os pares ordenados (0, 1), (1, 4), (2, 9), (3, 16), (4, 32) e (5, 64). Na função `plt.scatter` definimos a cor e a representação dos marcadores dos pontos em formato circular.

Para representar um conjunto de pontos com um gráfico 3D utilizamos o seguinte código em Python (em C++ o código fica similar):

```
import matplotlib.pyplot as plt

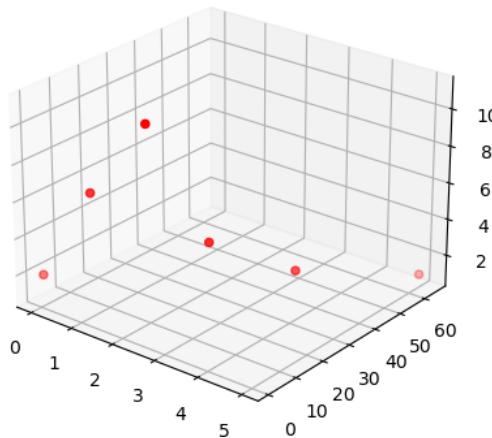
x = [0, 1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 32, 64]
z = [2, 7, 11, 5, 3, 1]

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

ax.scatter(x, y, z, color = 'r', marker = 'o')

plt.show()
```

Neste código, precisamos criar um gráfico auxiliar (ax) com a projeção em 3D. Além disso, criamos uma referência para adicionar ao elemento `figure` um subplot: `fig.add_subplot`.



A **Forma**, também chamada de marcador é outro tipo de variável visual, tal como: pontos, linhas, áreas, volumes e outros componentes. Esta variável visual engloba qualquer primitiva gráfica usada para representar os dados, incluindo símbolos, letras e palavras. Os principais marcadores usados na biblioteca gráfica `matplotlib` são:

'o' círculo, '*' estrela, '+' cruz, 'P' cruz com linhas mais grossas, 's' quadrado, 'D' losango, 'd' losango estreito, 'p' pentágono, 'h' hexágono, 'v' triângulo para baixo, '^' triângulo para cima, '<' triângulo para a esquerda, '>' triângulo para a direita, 'x' com linhas finas, 'X' com linhas mais grossas, '|' linha vertical e '_' linha horizontal.



Quando o conjunto de dados é representado apenas com marcadores, não devemos considerar diferenças em tamanhos, tonalização e orientação. Devemos analisar se um marcador se diferencia dos outros e se os marcadores têm áreas e formatos similares. Em uma única visualização podemos usar vários marcadores para diferenciar os agrupamentos de dados.

A escolha das **Cores** que serão usadas na visualização de dados ou objetos é uma etapa importante de um projeto. O conjunto de cores é outra importante variável visual, e seu uso correto permite melhorar a legibilidade de informações, gerar imagens realistas e focar a atenção do usuário.

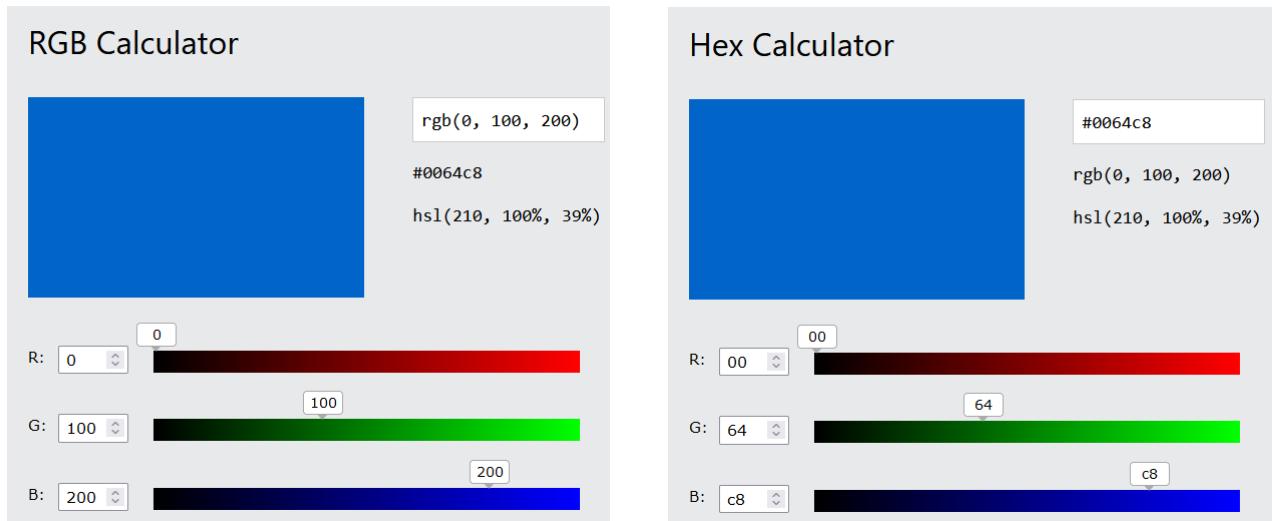
O modelo usado em telas de smartphones, monitores de vídeo e TV usa a geração de cores de forma aditiva. Desta forma, produzimos a cor de um elemento por meio da mistura de medidas das chamadas ondas luminosas, provocando a sensação de cor quando olhamos as imagens. Neste processo, a cor preta é gerada pela ausência de cor (nenhuma onda luminosa é transmitida) e o branco é a mistura de todas as cores (luminosidade total).

O modelo **RGB** define as cores usando as componentes na seguinte ordem: **vermelho (R)**, **verde (G)** e **azul (B)**. Os valores destas componentes podem ser medidos no intervalo [0, 255] ou em percentuais. Na biblioteca

matplotlib as cores podem ser definidas com percentuais, no intervalo [0, 1]. Logo, podemos definir a cor vermelha do gráfico 3D mostrado anteriormente da seguinte forma:

```
ax.scatter(x, y, z, color = (1, 0, 0), marker = 'o')
```

Utilizando um gerador de cores, podemos fazer as combinações de cores para usarmos nas representações gráficas. Estes geradores de cores sempre apresentam outras maneiras de representar as cores. Outro formato de cores bastante utilizado é o **Hexadecimal** com as cores representadas da seguinte forma: #RRGGBB. O intervalo representado neste formato de cor está entre 00 e FF, que indica a intensidade de cada cor.

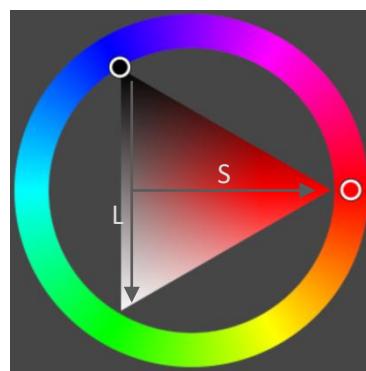


Disponível em: https://www.w3schools.com/colors/colors_rgb.asp

A cor RGB(0, 100, 200) é equivalente no formato hexadecimal a #0064c8. O código hexadecimal para esta cor pode ser definido da seguinte maneira:

```
ax.scatter(x, y, z, color = '#0064c8', marker = 'o')
```

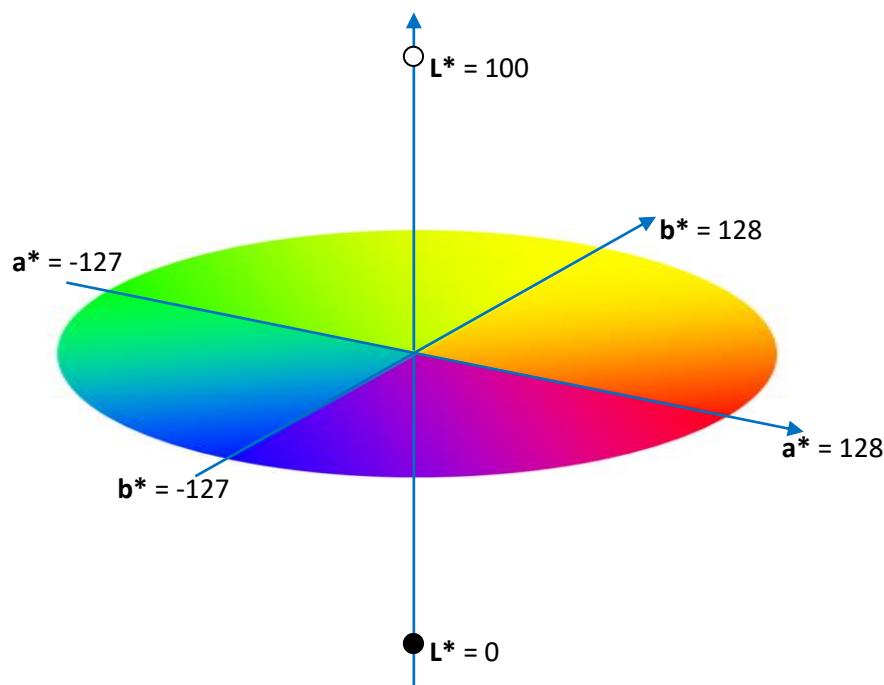
O formato **HSL** não tem suporte para todos os navegadores WEB e não existe a função de atribuição direta desta cor na biblioteca **matplotlib**. Neste formato, temos as seguintes componentes: **hue** – matiz (H), **saturação** (S) e **lightness** – brilho (L). A matiz é o ângulo no círculo de cores representadas entre 0 e 360°. Neste círculo, 0° é o vermelho, 60° é o amarelo, 120° representa o verde, 180° é o ciano, 240° representa a cor azul e 300° é a cor magenta. A saturação é o percentual de sombra de cinza sobre a cor (quanto maior o percentual, mais ‘pura’ é a cor). O brilho tem representação em percentual, indicando 0% para preto e 100% para branco.



O espaço de cores CIELAB, também conhecido como **L*a*b***, foi definido pela Comissão Internacional de Iluminação em 1976. Neste sistema, a cor pode ser definida por meio de três valores: L* para luminosidade perceptiva e a* e b* para as quatro cores únicas da visão humana: vermelho, verde, azul e amarelo. O CIELAB foi concebido como um espaço perceptivelmente uniforme, onde uma determinada mudança numérica corresponde a uma mudança de cor percebida de maneira semelhante. Este formato tem bastante utilidade na indústria para detectar pequenas diferenças de cor.

O espaço de cores CIELAB é um modelo de observador padrão, que independe de dispositivo. As cores que ele define não são relativas a um dispositivo específico, como um monitor de computador ou uma impressora, mas estão relacionadas ao observador padrão, que é uma média dos resultados de experimentos de correspondência de cores em condições de laboratório.

O espaço CIELAB é tridimensional e abrange toda a gama de percepção humana de cores. Este espaço tem base no modelo de cores oponentes da visão humana, onde vermelho e verde formam um par oponente, e azul e amarelo formam o outro par oponente. O valor de luminosidade, L^* , também conhecido como *Lstar*, define preto em 0 e branco em 100. O eixo a^* é relativo às cores adversárias verde-vermelha, com valores negativos em relação ao verde e valores positivos em relação ao vermelho. O eixo b^* representa os oponentes azul-amarelo, com números negativos para o azul e positivos para o amarelo.



Os eixos a^* e b^* são ilimitados e, dependendo do valor branco usado como referência, podem facilmente exceder ± 150 para cobrir a gama de visão humana. Entretanto, as implementações de software geralmente limitam esses valores por razões práticas. Os valores comuns fixados para a^* e b^* pertencem ao intervalo [-127, 128].

O valor de luminosidade, L^* no modelo proposto CIELAB é calculado usando a raiz cúbica da luminância relativa com um deslocamento próximo ao preto. Isso resulta em uma curva de potência efetiva com um expoente de aproximadamente 0,43, que representa a resposta do olho humano à luz sob condições de luz do dia (fotóptica). Este modelo pode ser representado como uma esfera (com os valores L^* representando polos), um cilindro (com os valores L^* representando os centros das bases) ou de um cone (com $L^* = 0$ representando o vértice e $L^* = 100$ representando o centro da base).

Existe um conjunto de 140 cores que são definidas por meio de seus respectivos nomes padronizados. Este conjunto de cores têm suporte em todos os navegadores web e na biblioteca de visualização do Python que estamos utilizando. No endereço mostrado a seguir, temos os nomes destas cores que podemos utilizar em nossos projetos de Visualização Científica:

https://www.w3schools.com/colors/colors_names.asp

As linhas são os objetos com 1 dimensão e são polilinhas (conjunto de segmentos) ou curvas. Para representar as linhas ou curvas com a biblioteca `matplotlib`, podemos usar a função `plot` informando um determinado conjunto de pontos ou o intervalo que deve ser representado. Neste primeiro exemplo, definimos os conjuntos de pontos por meio de suas coordenadas x e y. Para rotular cada ponto do gráfico, utilizamos o vetor de rótulos e a função `annotate` para atribuir para cada vértice seu respectivo rótulo.

```
import matplotlib.pyplot as plt
```

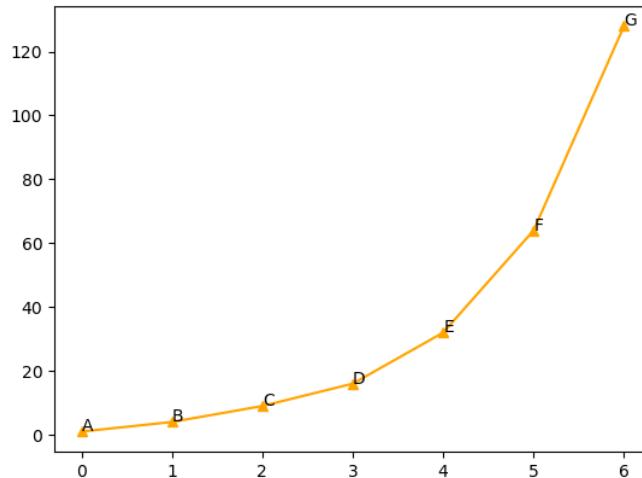
```

x = [0, 1, 2, 3, 4, 5, 6]
y = [1, 4, 9, 16, 32, 64, 128]
rotulos = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

for i, txt in enumerate(rotulos):
    plt.annotate(txt, (x[i], y[i]))

plt.plot(x, y, color = 'orange', marker = '^', linestyle = '-')
plt.show()

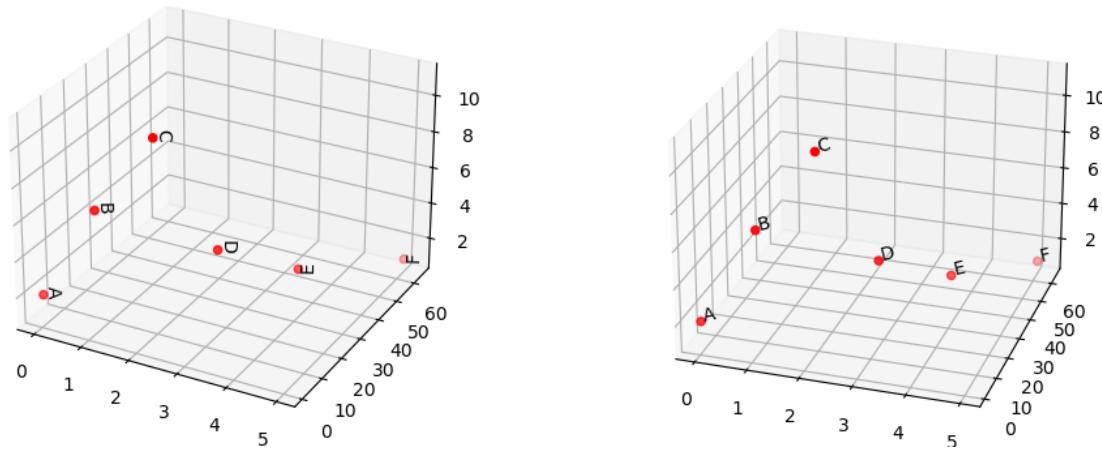
```



Se usarmos a cor azul, podemos simplificar as propriedades desta função `color = 'blue'`, `marker = '^'`, `linestyle = '-'` pela abreviação: `'b^-'`.

```
plt.plot(x, y, 'b^-')
```

O mesmo ocorre com as seguintes cores básicas: g – green; r – red; c – cyan; m – magenta; y – yellow; k – black; e w – white. Podemos colocar rótulos nos gráficos 3D usando a função similar `text3D`. Nesta função, devemos indicar a direção da representação de rótulos no atributo `zdir`, que pode ser um eixo (x, y ou z) ou um vetor.



```

import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

x = [0, 1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 32, 64]
z = [2, 7, 11, 5, 3, 1]
rotulos = ['A', 'B', 'C', 'D', 'E', 'F']

ax.scatter(x, y, z, color = 'r', marker = 'o')

```

```

for x, y, z, tag in zip(x, y, z, rotulos):
    label = tag
    ax.text3D(x, y, z, label, zdir = 'z')

plt.show()

```

Para representar mais polilinhas ou curvas, basta repetir a função `plot` com os dados que devem ser representados. Utilize os comandos da função `plot` mostrada a seguir para verificar como fica a representação gráfica dos dois conjuntos de pontos.

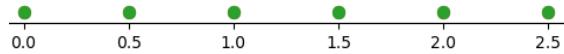
```

plt.plot(x, y, 'b^-')
plt.plot(x, x, 'gs-.')

```

Usando o mesmo raciocínio, podemos construir os gráficos com curvas com as bibliotecas **matplotlib** e **numpy**. Primeiro, definimos o intervalo que será representado por meio da função `arange`; depois, basta usar a função `plot` com as funções escolhidas. Podemos representá-las e utilizar legendas no gráfico.

A função `arange` cria um intervalo $[a, b]$ de números reais com o espaçamento c . O intervalo determinado por `arange(0, 3, 0.5)` é o seguinte:



No gráfico mostrado a seguir, vamos utilizar um espaçamento de 0.1 do intervalo $[0, 5]$:

```

import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 5, 0.1)

plt.plot(x, x, 'b--', label = 'y = x')
plt.plot(x, 2*x+1, 'g-', label = 'y = 2x + 1')
plt.plot(x, x**2+2*x+3, 'r-.', label = 'y = x^2 + 2x + 3')

plt.show()
plt.legend()

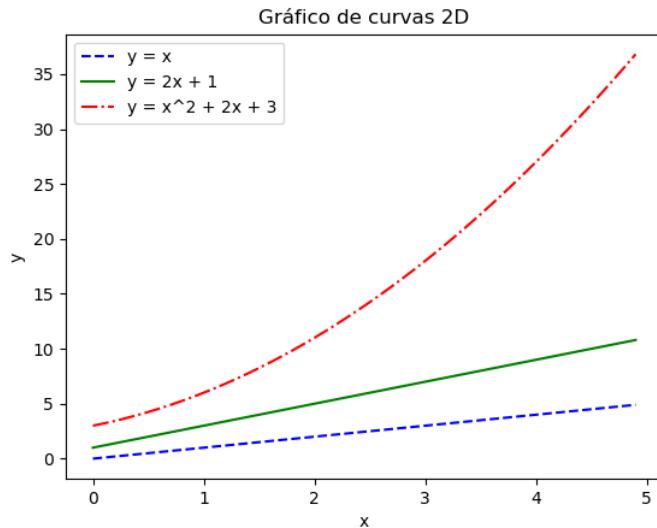
```

As legendas dos eixos e do gráfico são feitas com as seguintes propriedades:

```

plt.xlabel('x')
plt.ylabel('y')
plt.title('Gráfico de curvas 2D')

```



Note que não utilizamos os marcadores para representar as curvas. Caso seja necessário, basta inserí-los nas funções plot:

```
plt.plot(x, x, 'bo--', label = 'y = x')
plt.plot(x, 2*x+1, 'g^-', label = 'y = 2x + 1')
plt.plot(x, x**2+2*x+3, 'rs-.', label = 'y = x^2 + 2x + 3')
```

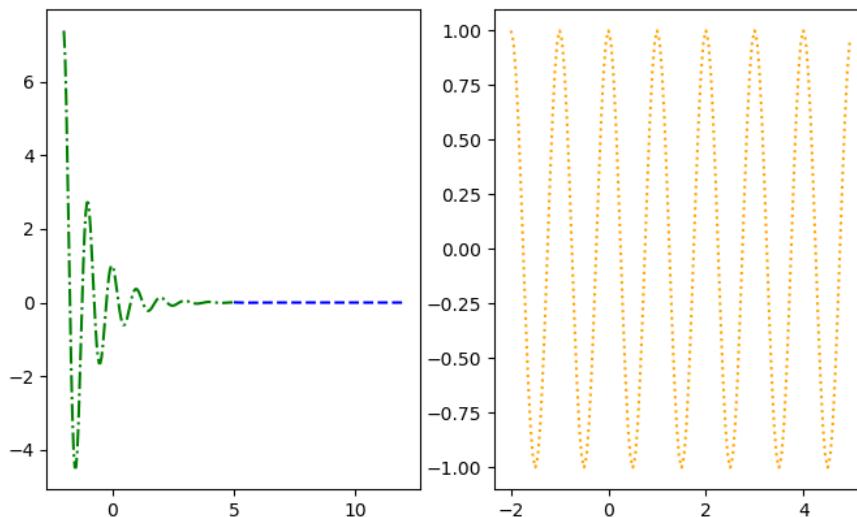
A escala fica ajustada automaticamente para a janela do gráfico. Para deixar os eixos x e y com a mesma escala, utilizamos a seguinte função:

```
plt.gca().set_aspect('equal', adjustable = 'box')
```

Se ajustarmos o intervalo com valores maiores, temos a definição da curva como uma polilinha. Veja o que acontece com a função quadrática do exemplo mostrado quando definirmos o conjunto x da seguinte maneira:

```
x = np.arange(0, 5, 1)
```

Podemos construir gráficos separados, que aparecem na mesma janela de visualização por meio da função subplot. Neste caso, definimos quantos gráficos aparecem em linha, em coluna, e qual é a ordem de cada gráfico. Por exemplo, subplot(2,3,1) significa que teremos 2 linhas de gráficos, 3 colunas de gráficos, e o gráfico atual é o primeiro da linha. No exemplo mostrado a seguir, temos 2 gráficos colocados lado a lado, ou seja, temos 1 linha e 2 colunas de gráficos. Podemos usar a definição de funções em separado ou definí-las no comando plot.



```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return np.exp(-x) * np.cos(2*np.pi*x)

x1 = np.arange(5, 12, 0.05)
x2 = np.arange(-2, 5, 0.05)

plt.figure()
plt.subplot(121)
plt.plot(x1, f(x1), 'b--', x2, f(x2), 'g-.')

plt.subplot(122)
plt.plot(x2, np.cos(2*np.pi*x2), color = 'orange', linestyle = ':')
plt.show()
```

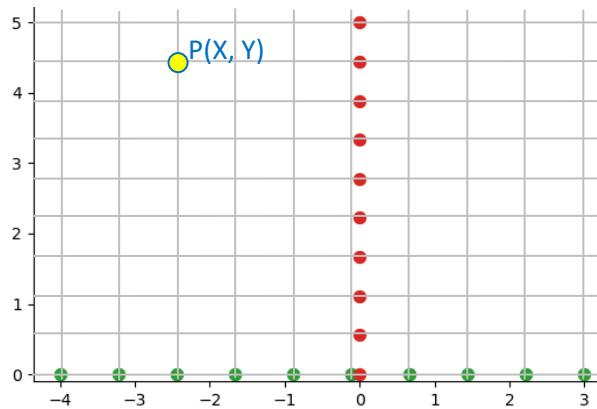
Agora modifique os gráficos, criando um terceiro gráfico, com a configuração de apresentação em 3 linhas e uma coluna; coloque legendas nos eixos e nos gráficos também.

A representação de linhas em 3D segue o mesmo raciocínio que vimos na representação de pontos em 3D. Nestes casos utilizamos a função `linspace` para definir os domínios das variáveis, indicando seus limites e a quantidade de pontos da amostra de cada domínio (indicada na variável `int`).

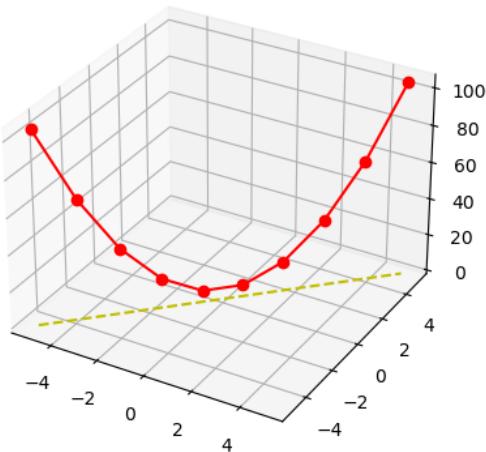
Para criar os pontos de um intervalo $[a, b]$ com a função `linspace`, devemos informar a quantidade `c` de pontos que este intervalo contém. Enquanto na função `arange` o conjunto de pontos do intervalo $[a, b]$ é definido a partir do incremento entre cada ponto, na função `linspace` informamos a quantidade de pontos do intervalo fechado $[a, b]$. Por exemplo, a função `linspace(-4, 3, 10)` cria 10 pontos no intervalo $[-4, 3]$.



Quando criamos `linspace` ou `arange` para duas ou três variáveis, conseguimos determinar uma estrutura geométrica de grade (grid) para o conjunto de dados ou variáveis. Trata-se de um modo eficiente para locação de pontos no plano ou no espaço. Desta forma, existe uma conectividade implícita dos pontos na grade determinada como locações de pontos em células.



Veja o exemplo da curva $z^2 + 5$ com $x = y$ e os intervalos indicados no código a seguir, com a formação de uma grade com as coordenadas x e y .



```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection = '3d')

int = 10
x = np.linspace(-5, 5, int)
y = np.linspace(-5, 5, int)
z = np.linspace(-10, 10, int)

ax.plot(x, y, z**2+5, 'ro-')
ax.plot(x, y, 0, 'y--')
plt.show()
```

Note que a função tracejada foi construída com a variável $z = 0$, ou seja, representa a projeção da curva sobre o plano xy . Agora modifique os intervalos das variáveis e a curva para novas representações em 3D.

Podemos representar curvas paramétricas com a função `plot` referenciando as variáveis dependentes e criando o domínio de uma das variáveis. No exemplo mostrado a seguir, temos a representação de uma hélice cilíndrica, com raio 5, definida com o intervalo indicado na função `linspace`.

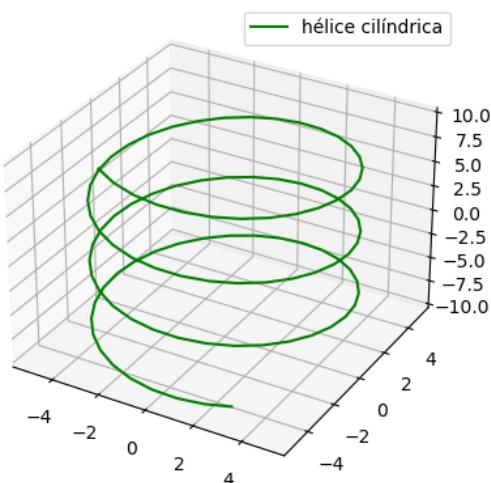
```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection = '3d')

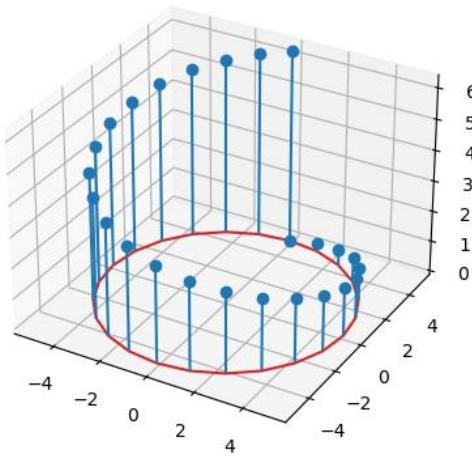
d = 5
z = np.linspace(-10, 10, 100)
x = d * np.sin(z)
y = d * np.cos(z)

ax.plot(x, y, z, 'g-', label = 'hélice cilíndrica')
ax.legend()

plt.show()
```



Modifique os parâmetros para criar novas hélices cilíndricas. Se utilizarmos a projeção da curva com $z = 0$, temos a representação mostrada anteriormente. A função `stem` cria os segmentos projetantes em um dos planos do sistema de representação 3D. Considere o exemplo de uma volta da hélice com diâmetro 5.

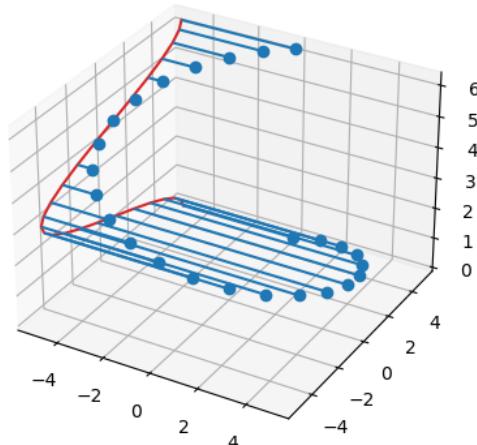


```
d = 5
z = np.linspace(0, 2*np.pi, 25)
x = d * np.sin(z)
y = d * np.cos(z)
```

```
ax.stem(x, y, z)
plt.show()
```

Usando a orientação padrão, temos a projeção ortogonal da curva sobre o plano xy. Para construir a projeção ortogonal sobre o plano yz, basta modificar a orientação da projeção e indicar a medida da coordenada da base (bottom).

```
ax.stem(x, y, z, bottom = -5, orientation = 'x')
```



Modifique os parâmetros para criar novas projeções da hélice cilíndrica.

Atividade 2

- 2.1. Crie outras representações gráficas de hélices cilíndricas com o comando `plot`.
- 2.2. Crie as representações gráficas de hélices cônicas por meio do mesmo comando `plot`.
- 2.3. Utilizando o comando `stem`, crie representações de hélices cônicas com orientações nos eixos x, y e z. Escolha a melhor posição do atributo `bottom` para definir a projeção de cada hélice cônica.

3. Fundamentos dos dados

Cada observação ou variável que representa uma instância de dado fornece uma única informação, que pode ser categorizada como ordinal (numérica) ou nominal (não numérica).

Os dados ordinais podem ser classificados em:

- **binários** – quando podem assumir dois valores (em geral, 0 ou 1);
- **discretos** – quando assumem um valor inteiro pertencente a um conjunto específico (por exemplo, {0, 1, 2, 3, 5, 9, 12, 18}); ou
- **contínuos** – quando assumem valores reais.

Os dados nominais podem ser classificados da seguinte maneira:

- **categóricos** – quando possuem um valor pertencente a um conjunto finito de possibilidades (conjunto de cores - azul, amarelo, verde ou branco);
- **ranqueados** – quando representam uma variável categórica com alguma ordem (tamanhos - pequeno, médio ou grande); ou
- **arbitrários** – quando as variáveis possuem muitas possibilidades de valores, sem relação de ordem específica (endereços, cidades, nomes completos).

Nos exemplos de conjuntos de dados mostrados a seguir, temos os tipos de dados ordinais e nominais:

Estudantes

ID	Nome	Período	Curso	Idade	Índice acadêmico	Tipo sanguíneo
21455	Carlos	1	Engenharia Florestal	19	78,65	O+
21456	Maria	5	Matemática	23	68,45	A-
21457	Inês	3	Medicina	20	86,74	AB+
21458	Eduardo	3	Engenharia Química	22	59,32	AB-
21459	Juliana	7	Direito	25	88,74	B+

Podemos categorizar os dados utilizando o conceito de escala. Neste caso, os atributos que definem as variáveis podem ser categorizados como uma relação de ordem, uma métrica de distância ou a existência de zero absoluto. Na relação de ordem, os dados podem ser ordenados de acordo com os valores de uma variável, permitindo que os dados sejam ranqueados.

No exemplo de dados dos estudantes, o conjunto está ordenado segundo a variável ID; porém, podemos ranquear os dados usando o índice acadêmico (do maior para o menor ou vice-versa) ou o nome de cada estudante (ordem alfabética). A métrica de distância permite o cálculo de distâncias entre os registros ordinais, não sendo possível fazer o mesmo com variáveis nominais. Quando for possível, podemos definir o zero absoluto do conjunto de dados, permitindo fixar um menor valor. Desta forma, fica fácil diferenciar tipos de variáveis ordinais do conjunto.

Os conjuntos de dados possuem estruturas relacionadas às formas de representação e aos tipos de relacionamentos entre instâncias. Estas estruturas podem ser classificadas em: escalares, vetores, tensores, geométricas e grades.

As estruturas escalares são foco de análise para a visualização de dados. Cada escalar representa um único número em uma instância dos dados (por exemplo, idade, índice de preços ou índice acadêmico).

Nas situações em que as dimensões dos dados excedem as capacidades das técnicas de análise de dados, aplicamos a técnica denominada **Redução de Dimensionalidade**. Desta forma, temos que investigar meios de reduzir a dimensionalidade dos dados, tentando preservar o máximo possível das informações contidas nestes dados.

Uma maneira bastante conhecida para determinar quais variáveis devem ser escolhidas é a Análise de Componentes Principais (PCA – Principal Component Analysis). Com o uso desta técnica, criamos um novo conjunto de variáveis (chamadas de componentes principais) a partir de combinações lineares das já existentes. A técnica PCA utiliza as técnicas da álgebra linear para definir os componentes principais de uma maneira que descreva a maior variação possível dos dados originais com menos variáveis novas. Isso acontece sequencialmente: o primeiro componente principal descreve a maior variação possível; o segundo descreve o máximo possível da variação remanescente (com a restrição de que não deve ser correlacionada com o primeiro componente principal); e assim por diante. No máximo, pode haver tantos componentes principais quanto variáveis originais, mas podemos optar por usar menos deles se observarmos uma grande porcentagem de variação capturada em um subconjunto.

Como os componentes principais devem capturar a variação entre as variáveis existentes, devemos padronizar todas as variáveis antes de utilizar o PCA, deixando todos os dados na mesma escala. Após a conclusão da padronização dos dados, podemos utilizar o PCA. Temos opções de utilizar as bibliotecas do Python com o PCA, ou podemos programar a técnica PCA, que tem basicamente os seguintes passos:

1. Considere o conjunto de dados $(x_i), i = 1, \dots, n, \in \mathbb{R}^k$.
2. Determine o centróide do conjunto de dados: $c = \frac{1}{n} \sum_i x_i$.
3. Encontre a matriz de covariância: $A_{jl} = \frac{1}{n} \sum_i (x_{ij} - c_j)(x_{il} - c_l)$.
4. Os autovetores da matriz de covariância formam o sistema de coordenadas dependentes do conjunto de dados. Se colocarmos os autovetores em ordem decrescente, usando como critério os respectivos autovalores, os primeiros m autovetores formam as principais dimensões do conjunto de dados.

Vamos utilizar com maior frequência dois exemplos de conjuntos de dados com o uso da biblioteca matplotlib:

- **Iris**, que usa as variáveis comprimento de sépala, comprimento de pétala, largura de sépala e comprimento de sépala (todas em cm), para fazer a classificação das flores em 3 espécies: Iris Setosa, Iris Versicolor e Iris Virginica (disponível em: <https://archive.ics.uci.edu/ml/datasets/iris>); e

- **Penguins**, que usa as variáveis ilha, comprimento do bico, profundidade do bico, comprimento da nadadeira, massa corporal e ano para classificar os pinguins em 3 espécies: Adelie, Gentoo e Chinstrap (disponível em: https://inria.github.io/scikit-learn-mooc/python_scripts/trees_dataset.html).

Outros conjuntos de dados podem ser encontrados na seguinte página: <https://www.maptive.com/free-data-visualization-data-sets/>

Os arquivos csv com os conjuntos de dados têm os seguintes formatos:

```
Id,Comprimento da Sépala,Largura da Sépala,Comprimento da Pétala,Largura da Pétala,Espécie
1,5.1,3.5,1.4,0.2,Iris-setosa
2,4.9,3.0,1.4,0.2,Iris-setosa
3,4.7,3.2,1.3,0.2,Iris-setosa
4,4.6,3.1,1.5,0.2,Iris-setosa
5,5.0,3.6,1.4,0.2,Iris-setosa
...
150,5.9,3.0,5.1,1.8,Iris-virginica
```

```
Id,Espécie,Ilha,Comprimento do bico,Profundidade do bico,Comprimento da nadadeira,Massa
corporal,Sexo,Ano
1,Adelie,Torgersen,39.1,18.7,181,3750,male,2007
2,Adelie,Torgersen,39.5,17.4,186,3800,female,2007
3,Adelie,Torgersen,40.3,18,195,3250,female,2007
4,Adelie,Torgersen,NA,NA,NA,NA,NA,2007
5,Adelie,Torgersen,36.7,19.3,193,3450,female,2007
...
344,Chinstrap,Dream,50.2,18.7,198,3775,female,2009
```

Nestes conjuntos utilizados como exemplos, temos sempre mais de 3 atributos usados para fazer as classificações. Usando o conceito de redução de dimensionalidade, podemos analisar o melhor conjunto de atributos que pode ser usado para garantir as análises corretas das classificações destes dados.

No processo de visualização de dados, devemos analisar o gráfico construído para verificar se é possível perceber agrupamentos dos dados similares ou se existe um relacionamento entre os dados. Devemos analisar também se é possível categorizar cognitivamente os grupos de dados e examinar casos especiais que não ficaram agrupados ou que têm relações com os respectivos grupos.

A **Orientação** ou direção descreve como um marcador deve ser rotacionado em relação à respectiva variável de dados mapeada desta variável visual. Vale lembrar que nem todos os marcadores possibilitam o uso da orientação (por exemplo, círculos e asteriscos). Os melhores marcadores para usarmos a orientação são os que possuem apenas um eixo de simetria (por exemplo, segmentos, losangos ou triângulos alongados). Quando usamos a orientação com uma variação menor do que 180° , os gráficos apresentam resultados interessantes.



Vamos utilizar a biblioteca **matplotlib** para criar a representação 2D dos dados usando a variável visual de orientação. Além disso, vamos usar a função da biblioteca **pandas** para ler o arquivo csv com os dados. Com os dados do conjunto Iris, usaremos a orientação relacionada a uma das variáveis: a largura da sépala (vamos chamá-la de `incl`). Neste caso, precisamos criar uma lista de índices `i` para utilizar um loop que determina os ângulos de acordo com os valores das variáveis.

A inclinação no sentido horário é considerada com medida negativa do ângulo. Logo, faremos uma escala para que o menor valor da variável `incl` seja 0 e o maior valor seja a amplitude máxima -45. Neste caso, utilizamos o marcador de losango (código 2, 1).

Os vetores de rótulos e de cores controlam as mudanças de espécies do conjunto de dados. Neste caso, os dados precisam estar ordenados segundo o filtro de espécies para que o código funcione corretamente.

```

import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

iris = pd.read_csv('C:/dados/iris.csv')

incl = np.array(iris.loc[:, 'Largura da Sépala'])
x = np.array(iris.loc[:, 'Comprimento da Sépala'])
y = np.array(iris.loc[:, 'Comprimento da Pétala'])
z = np.array(iris.loc[:, 'Espécie'])

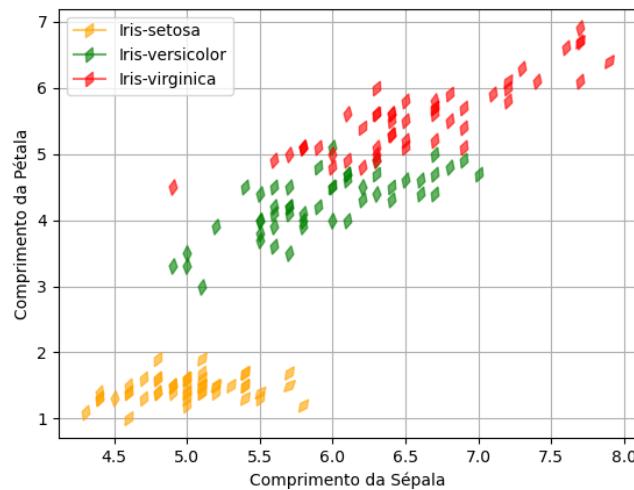
i = np.arange(0, len(incl), 1)
j = (incl - min(incl))/(max(incl) - min(incl))
w = -45*j

label = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
cor = ['orange', 'green', 'red']
j = 0

for k in i:
    marker = (2, 1, w[k])
    if z[k] == z[k-1]:
        plt.plot(x[k], y[k], marker = marker, markersize = 10, color = cor[j - 1], alpha = 0.6)
    else:
        j +=1
        plt.plot(x[k], y[k], marker = marker, markersize = 10, color = cor[j - 1], alpha = 0.6,
                  label = label[j - 1] )

plt.legend(scatterpoints = 1)
plt.xlabel('Comprimento da Sépala')
plt.ylabel('Comprimento da Pétala')
plt.grid()
plt.show()

```



Desta forma, temos o primeiro gráfico que mostram os dados por meio das variáveis visuais Cor e Orientação. Modifique as variáveis, cores e a amplitude máxima do ângulo para construir novos gráficos. Crie diferentes marcadores para cada espécie.

A **Textura** pode ser considerada como a combinação das seguintes variáveis visuais: forma, cor e orientação. Variando-se as configurações destes elementos, obtemos novas texturas. Podemos escolher as seguintes texturas para marcadores ou polígonos com a biblioteca matplotlib: '-', '+', 'x', '\\', '*', 'o', 'O', '.', '/'.



No exemplo do conjunto de dados Iris, os vetores de texturas e de marcadores são usados no loop que controla as mudanças de espécies do vetor z. Como foi mostrado no exemplo anterior, este conjunto de dados está ordenado com o filtro de espécies para que as trocas de rótulos, de texturas e de rótulos funcionem corretamente.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

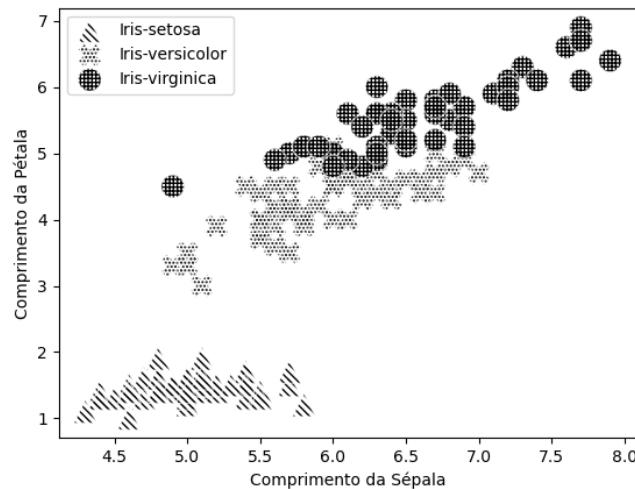
iris = pd.read_csv('C:/dados/iris.csv')

x = np.array(iris.loc[:, 'Comprimento da Sépala'])
y = np.array(iris.loc[:, 'Comprimento da Pétala'])
z = np.array(iris.loc[:, 'Espécie'])

i = np.arange(0, len(x), 1)
label = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
tex = ['\\', '.', '+']
marker = ['^', 'x', 'o']
j = 0

for k in i:
    if z[k] == z[k-1]:
        plt.scatter(x[k], y[k], marker = marker[j - 1], s = 200, facecolor = 'white',
                    hatch = 5*tex[j-1], alpha = 0.5)
    else:
        j += 1
        plt.scatter(x[k], y[k], marker = marker[j - 1], s = 200, facecolor = 'white',
                    hatch = 5*tex[j-1], alpha = 0.5, label = label[j-1])

plt.xlabel('Comprimento da Sépala')
plt.ylabel('Comprimento da Pétala')
plt.legend(scatterpoints=1)
plt.show()
```

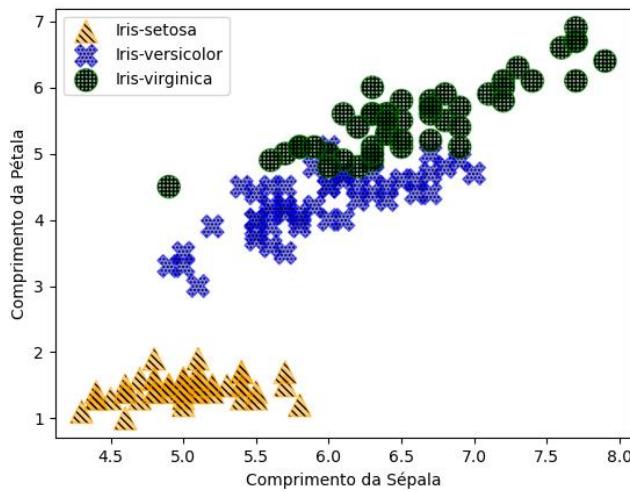


Criando-se um vetor de cores, podemos usá-lo como cor de fundo de cada marcador, melhorando a visualização deste conjunto de dados.

```
cor = ['orange', 'blue', 'green']

plt.scatter(x[k], y[k], marker = marker[j - 1], s = 200, facecolor = cor[j-1],
            hatch = 5*tex[j-1], alpha = 0.5)
```

Modifique variáveis, cores e estilos de texturas para criar novos gráficos deste conjunto de dados.

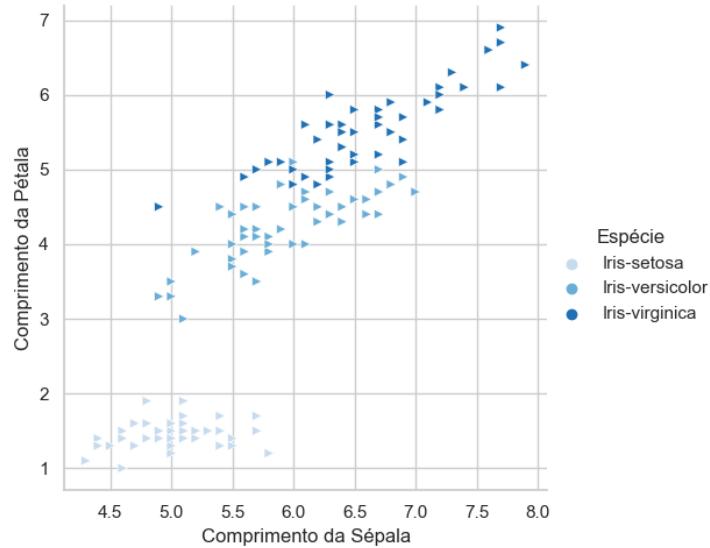


Agora vamos utilizar a biblioteca **seaborn**, que possui a estrutura matplotlib com interface para representações gráficas de dados estatísticos. Com esta biblioteca, podemos criar com poucos comandos as representações de dados separados por suas respectivas classes de forma automática por meio de cores distintas. Se for necessário, podemos programar a orientação de marcadores da mesma forma mostrada no exemplo anterior.

```
import pandas as pd
import seaborn as sns

iris = pd.read_csv('C:/dados/iris.csv')

sns.relplot(data = iris, x = 'Comprimento da Sélpa', y = 'Comprimento da Pétala',
            hue = 'Espécie', marker = '>', palette = 'Blues')
```



Modifique os parâmetros desta função para obter novas visualizações deste conjunto de dados.

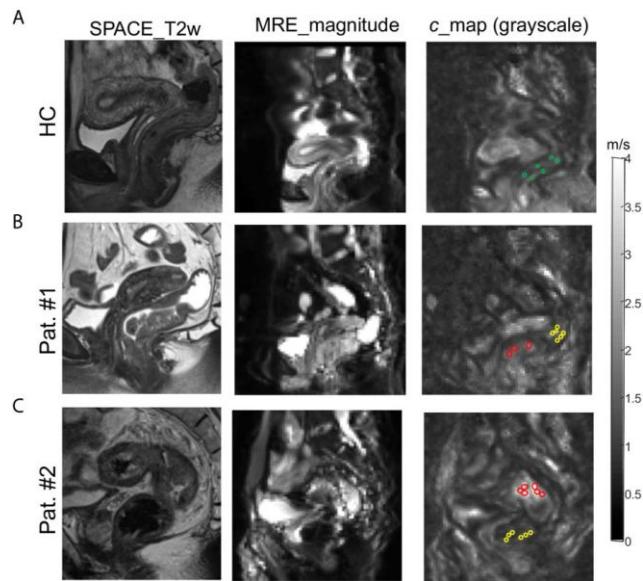
3.1. Mapas de cores

Um Mapa de Cor (**colormap**) ou Paleta de Cores (**palette**) pode ser utilizado para mostrar pontos críticos de gráficos ou melhorar a percepção da variação de valores dos dados. Por exemplo, em uma representação de superfície terrestre podemos usar cores quentes para regiões com baixas altitudes e cores frias para os pontos com maiores altitudes.

Podemos considerar que os Mapas de cores servem para: Imitar a realidade, mostrar classificações, mostrar valores, chamar a atenção do usuário para regiões do gráfico ou mostrar agrupamentos de dados. As representações gráficas do que vemos depende do dispositivo de exibição, pois os esquemas de cores aparecem diferentes em

monitores diferentes. A eficácia das escolhas de cores depende dos seguintes elementos: display, conjunto de dados, aplicativo e público alvo.

Algumas escalas de cores são chamadas de univariadas e representam caminhos por meio do espaço de cores ou uma parte do modelo de representação de cores. A ideia deste tipo de escala é variar um único componente de modelo de cor, mantendo-se os demais componentes constantes. Um exemplo muito utilizado é de escala de cinza.



Disponível em: <https://www.frontiersin.org/articles/10.3389/fonc.2021.701336/full>

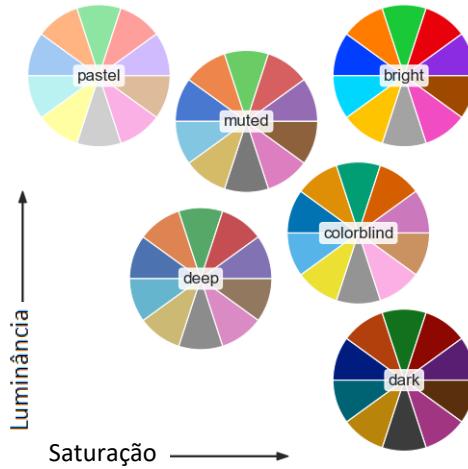
As escalas de cores redundantes possuem dois ou mais componentes de cores variados juntos. Nestes exemplos, temos a variação de cores com suavidade usando alguma medida de atributo como parâmetro, reforçando características dos dados e combinando características de escalas de cores simples.



Disponível em: <https://sawx.co.za/>

Na maioria das representações gráficas um mapa de cores perceptivelmente uniforme é a melhor escolha. Vamos utilizar os valores comparativos das variáveis L^* , a^* e b^* do CIELAB para definir os mapas de cores. Quando o valor da luminosidade L^* monotonicamente crescente (ou decrescente) através do mapa de cores, os dados serão interpretados de maneira mais adequada pelos usuários. Alguns valores L^* nos mapas de cores variam de 0 a 100 (binários e outros em tons de cinza), e outros começam em torno de $L^* = 20$. Os mapas com menor variação de L^* terão um alcance perceptual menor. A função varia também os valores dos tons de cores a^* e b^* de maneira linear em alguns casos.

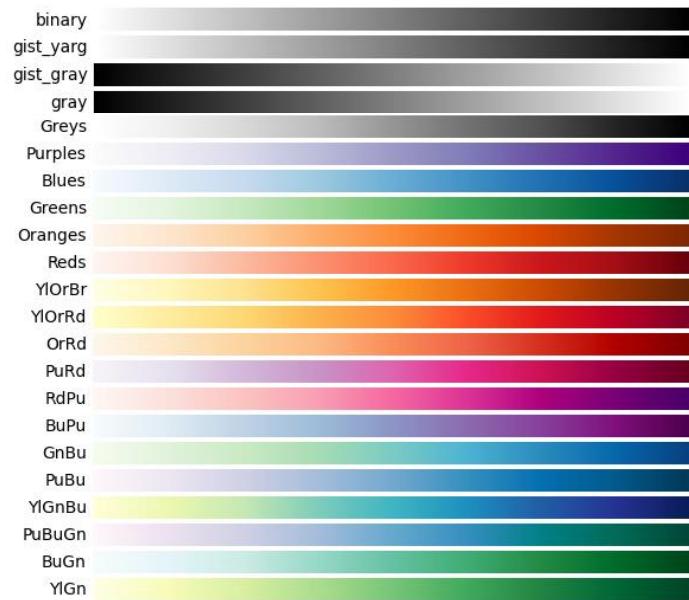
De acordo com a variação de luminância e de saturação, temos a variação de cores definida para os tons: pastel, muted (cor colocada em mudo: quando misturada com cinza ou branco apresenta tom suavizado), bright (brilhante), deep (profunda), colorblind (daltônica) e dark (escura).



Fonte: https://seaborn.pydata.org/tutorial/color_palettes.html

Podemos usar os conjuntos de mapas de cores sequenciais pré-determinados da biblioteca **matplotlib**, que são mostrados a seguir:

```
{'binary', 'gist_yarg', 'gist_gray', 'gray', 'Greys', 'Purples', 'Blues', 'Greens', 'Oranges',
'Reds', 'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu', 'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn',
'BuGn', 'YlGn'}.  
}
```



Fonte: <https://matplotlib.org/3.5.1/tutorials/colors/colormaps.html>

Na biblioteca **seaborn**, temos estes mapas de cores sequenciais que serão mostrados a seguir:



```
{'rocket', 'crest', 'mako', 'magma', 'flare', 'cubelix'}.
```

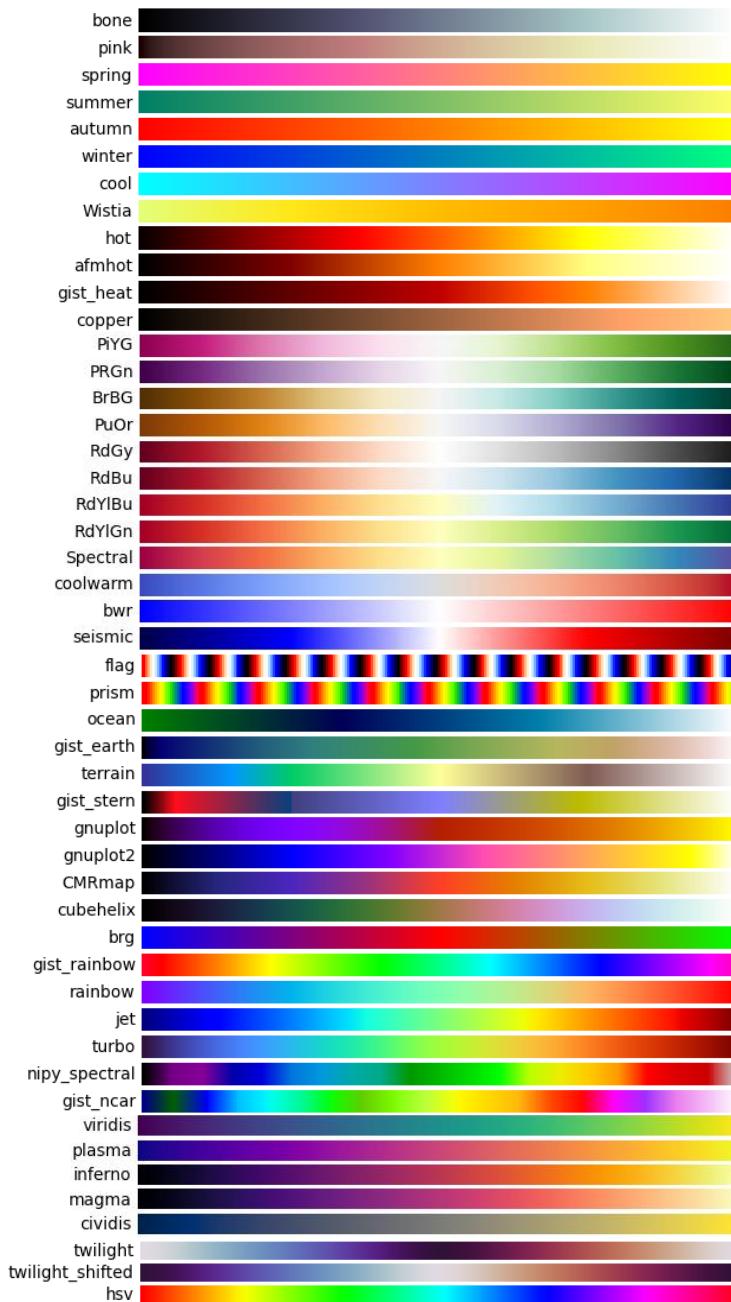
O mapa de cores chamado de escala dupla possui a variação de duas escalas distintas unidas no meio neutro. Nestes mapas, temos valores monotonicamente crescentes de L^* até um máximo, que deve ser próximo de 100, seguido de valores monotonicamente decrescentes de L^* . Este tipo de mapa de cor pode ser usado para segmentar valores em dois grupos (por exemplo, positivo × negativo), ou enfatizar os dois extremos do intervalo de dados.



Nos mapas de cores cílicos, queremos começar e terminar na mesma cor e encontrar um ponto central que define variações simétricas de cores. Nestes mapas, L^* deve mudar monotonicamente do início ao meio e inversamente do meio ao fim. Deve ser simétrico no lado crescente e decrescente, e diferir apenas nas tonalidades. Nas extremidades e no meio, o valor de L^* inverte a direção, que deve ser suavizada no espaço do mapa.

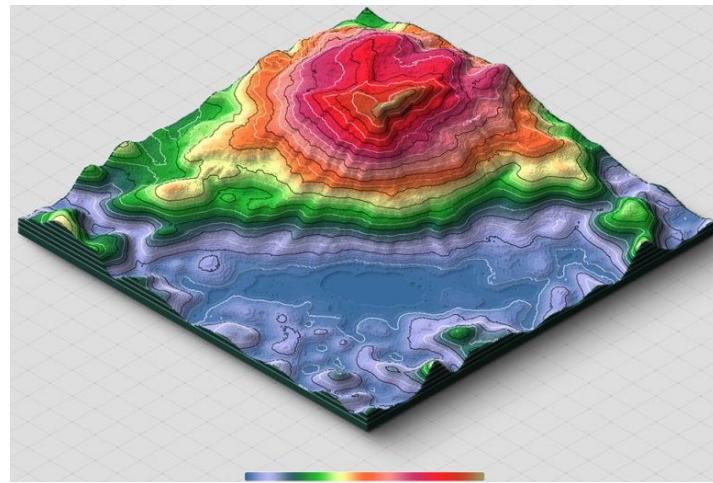
Outros mapas de cores pré-definidos estão mostrados a seguir. Nestes mapas, temos combinações de duas ou mais cores e alguns exemplos com repetições de padrões (cílicos).

```
{'bone', 'pink', 'spring', 'summer', 'autumn', 'winter', 'cool', 'Wistia', 'hot', 'afmhot',
'gist_heat', 'copper', 'PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu', 'RdYlBu', 'RdYlGn',
'Spectral', 'coolwarm', 'bwr', 'seismic', 'flag', 'prism', 'ocean', 'gist_earth', 'terrain',
'gist_stern', 'gnuplot', 'gnuplot2', 'CMRmap', 'cubehelix', 'brg', 'gist_rainbow', 'rainbow',
'jet', 'turbo', 'nipy_spectral', 'gist_ncar', 'viridis', 'plasma', 'inferno', 'magma',
'cividis', 'twilight', 'twilight_shifted', 'hsv'}.
```



Fonte: <https://matplotlib.org/3.5.1/tutorials/colors/colormaps.html>

Em modelos 3D, podemos usar um mapa com mais de duas cores para melhorar a visualização dos dados.



Disponível em: <https://www.deviantart.com/templay-team/art/3D-Map-Generator-Terrain-Height-Color-Map-693066368>

Usando-se posicionamento, marcadores e cores, podemos criar uma visualização de dados. Outra variável visual que pode ser utilizada é o **Tamanho**, que pode auxiliar nas interpretações dos dados exibidos. Neste caso, utilizamos um atributo para definir os tamanhos dos marcadores mostrados em 2D ou 3D.

O tamanho permite o mapeamento de intervalos de variáveis, possibilitando o incremento gradual em um determinado intervalo. Além disso, podemos mapear outra variável no gráfico usando como critério os tamanhos de marcadores. Os tamanhos de marcadores devem ser evitados quando a variável escolhida tem poucas variações de valores, pois impossibilita que o usuário identifique as diferenças entre os dados apresentados.

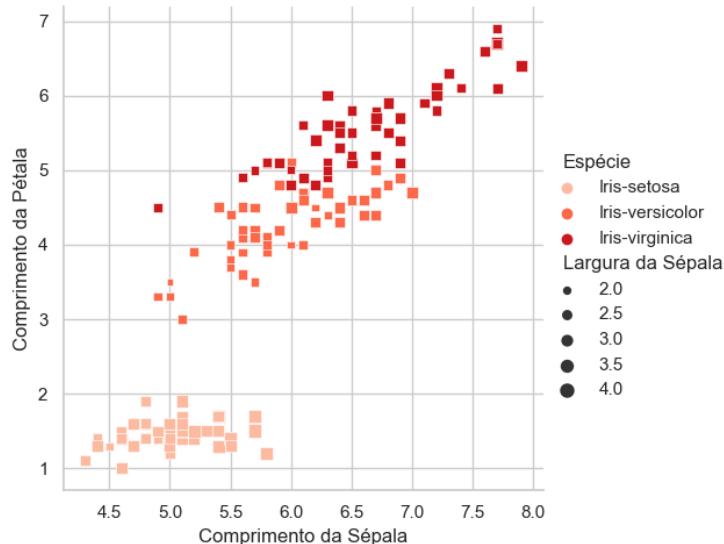


O atributo **size** pode ser usado para representar uma terceira variável no gráfico 2D do conjunto Iris. A variável de largura da sépala foi usada para mapear o tamanho de cada marcador.

```
import pandas as pd
import seaborn as sns

iris = pd.read_csv('C:/dados/iris.csv')

sns.relplot(data = iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
            hue = 'Espécie', marker = 's', palette = 'Reds', size = 'Largura da Sépala')
```

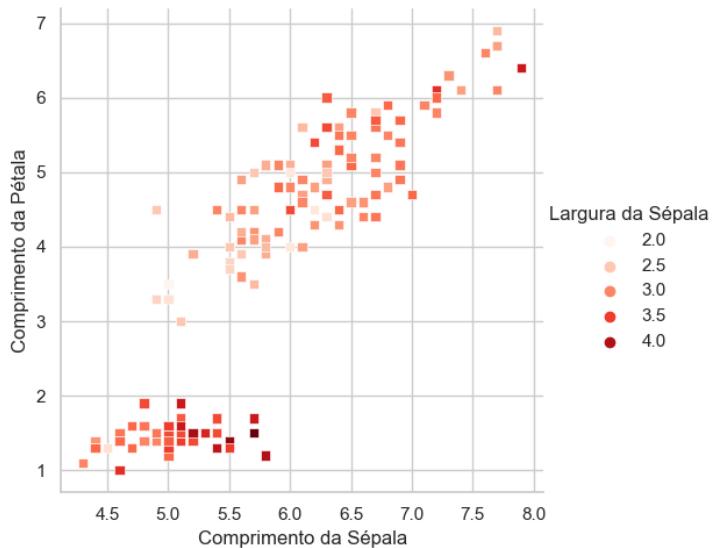


Modifique a variável usada como critério de tamanho neste exemplo de dados Iris.

Outra variável visual que pode ser utilizada nas representações gráficas é o **Brilho**. Com o uso do brilho, podemos representar a variação dos valores para outra variável do conjunto de dados. De forma automática, os dados são apresentados em uma escala do tipo gradiente, variando as cores de acordo com o valor de cada atributo escolhido para representar o brilho.

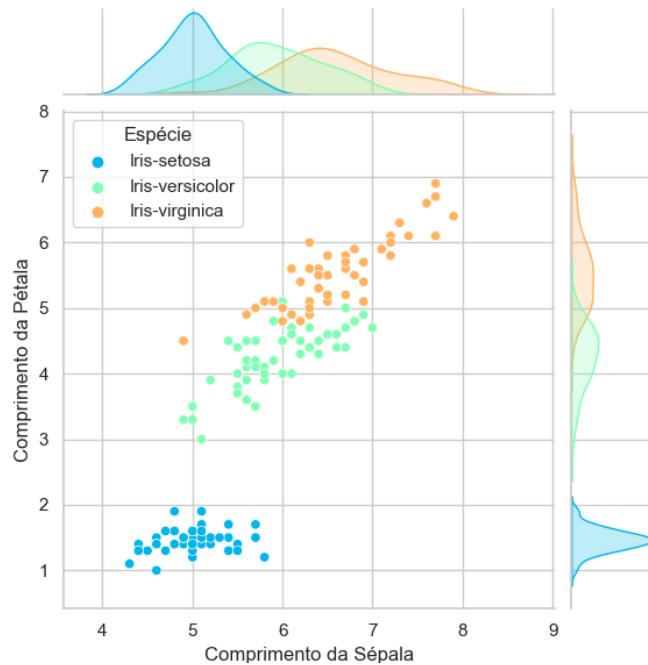
Se escolhermos a largura da sépala no conjunto de dados iris, com a variação em tons de vermelho, temos o seguinte gráfico:

```
sns.relplot(data = iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
            hue = 'Largura da Sépala', marker = 's', palette = 'Reds')
```



A função `jointplot` representa graficamente a distribuição conjunta entre duas variáveis com a distribuição marginal de cada variável.

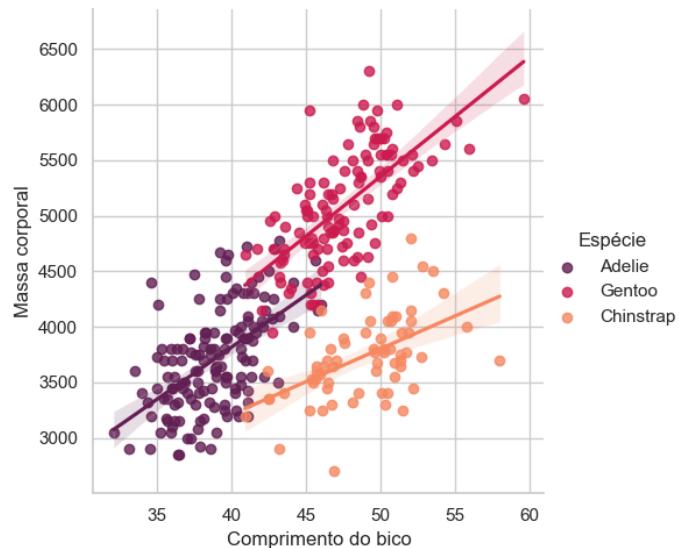
```
sns.jointplot(data = iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
               hue = 'Espécie', marker = 'o', palette = 'rainbow')
```



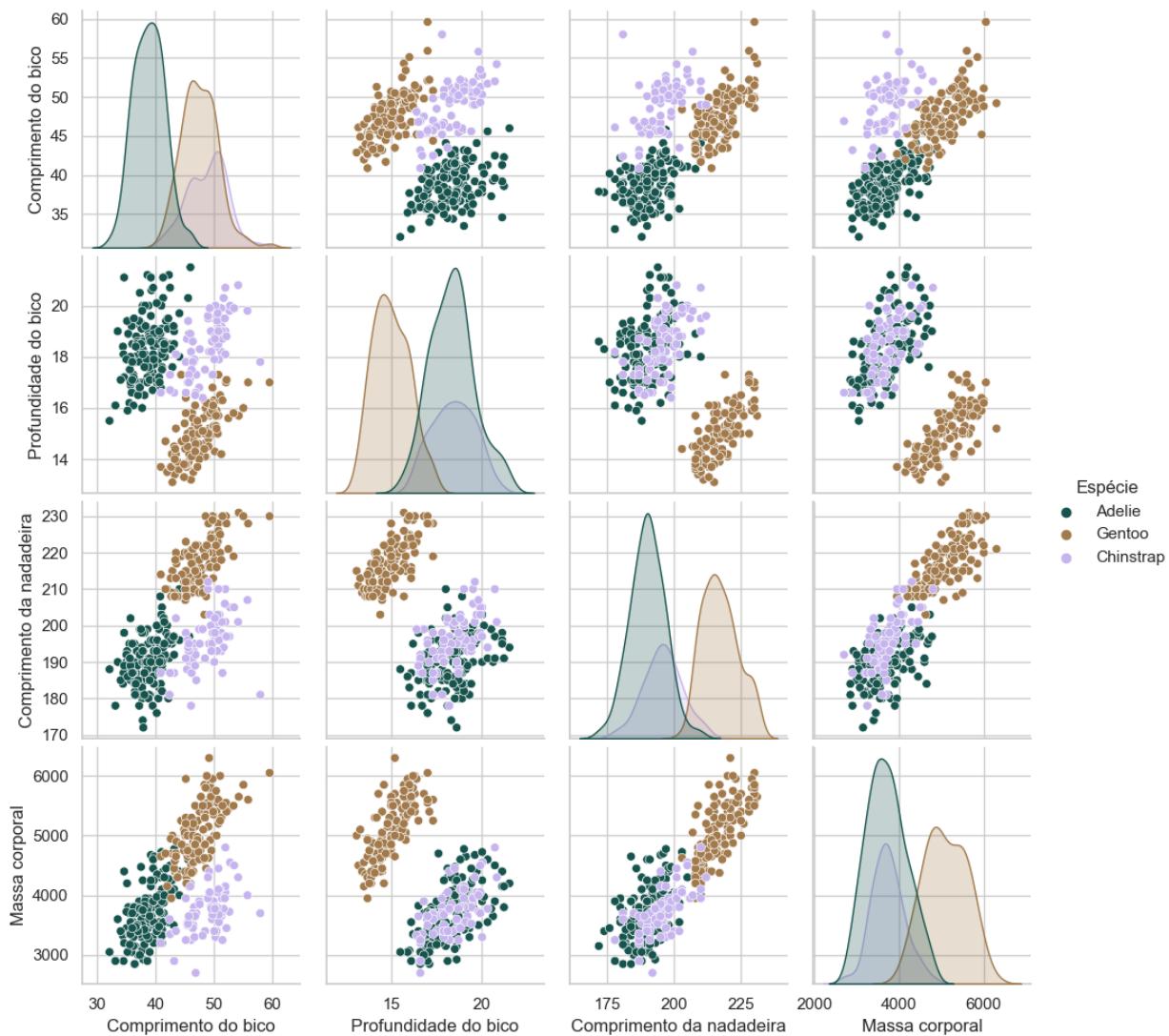
Utilize duas variáveis para fazer a representação 2D do conjunto de dados dos pinguins com as funções `relplot` e `joinplot`.

Com a função `lmplot`, podemos fazer a representação dos dados com a regressão linear múltipla, de acordo com cada espécie do conjunto. Os exemplos mostrados a seguir são do conjunto dos pinguins.

```
sns.lmplot(data = penguins, x = 'Comprimento do bico', y = 'Massa corporal',
            hue = 'Espécie', palette = 'rocket')
```



Para uma análise completa dos dados, podemos criar as combinações de variáveis em gráficos que ficam representados lado a lado, no formato de uma matriz.



Nestes casos, precisamos informar quais variáveis serão desconsideradas, para não afetar a análise desta representação.

```
penguins.drop(['Id', 'Ano'], inplace = True, axis = 1)
sns.pairplot(data = penguins, hue = 'Espécie', palette = 'cubebehelix')
```

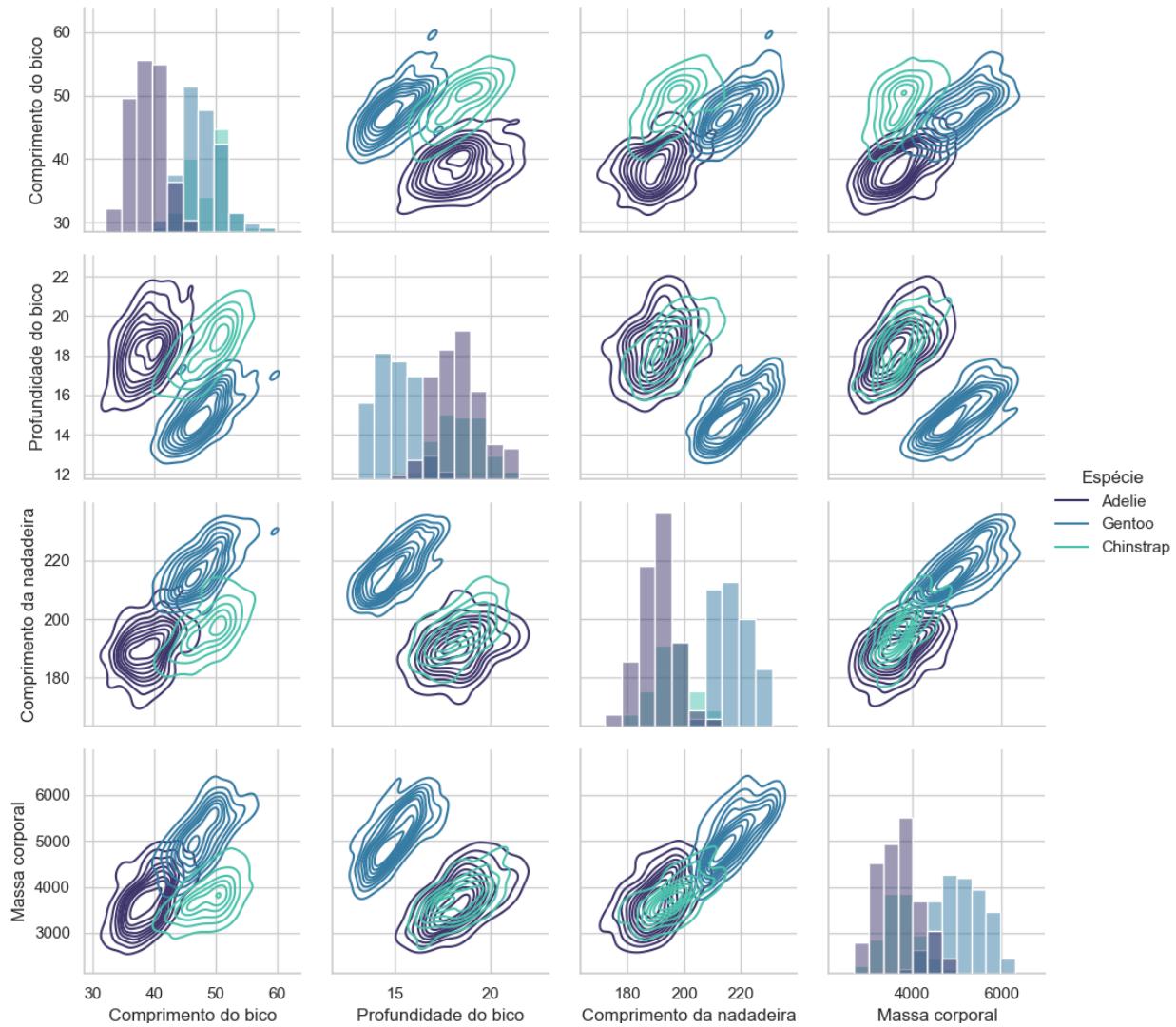
Para criar marcadores diferentes para cada espécie, basta usar o atributo `markers = ['D', 's', '^']` com 3 marcadores escolhidos.

Podemos utilizar o gráfico que mostra a estimativa de densidade do kernel (núcleo). Trata-se de um método para visualizar a distribuição de observações em um conjunto de dados parecido com o histograma, mas que utiliza as coordenadas dos pontos. O resultado fica parecido com as representações de curvas de nível de superfícies. A representação de kernel (kde) representa os dados usando uma curva de densidade de probabilidade contínua em uma ou mais dimensões.

```
sns.pairplot(data = penguins, hue = 'Espécie', palette = 'mako', kind = 'kde')
```

Para representar os gráficos da diagonal como histogramas de frequência, utilizamos o comando `PairGrid`.

```
g = sns.PairGrid(data = penguins, hue = 'Espécie', palette = 'mako')
g.map_diag(sns.histplot)
g.map_offdiag(sns.kdeplot)
g.add_legend()
```

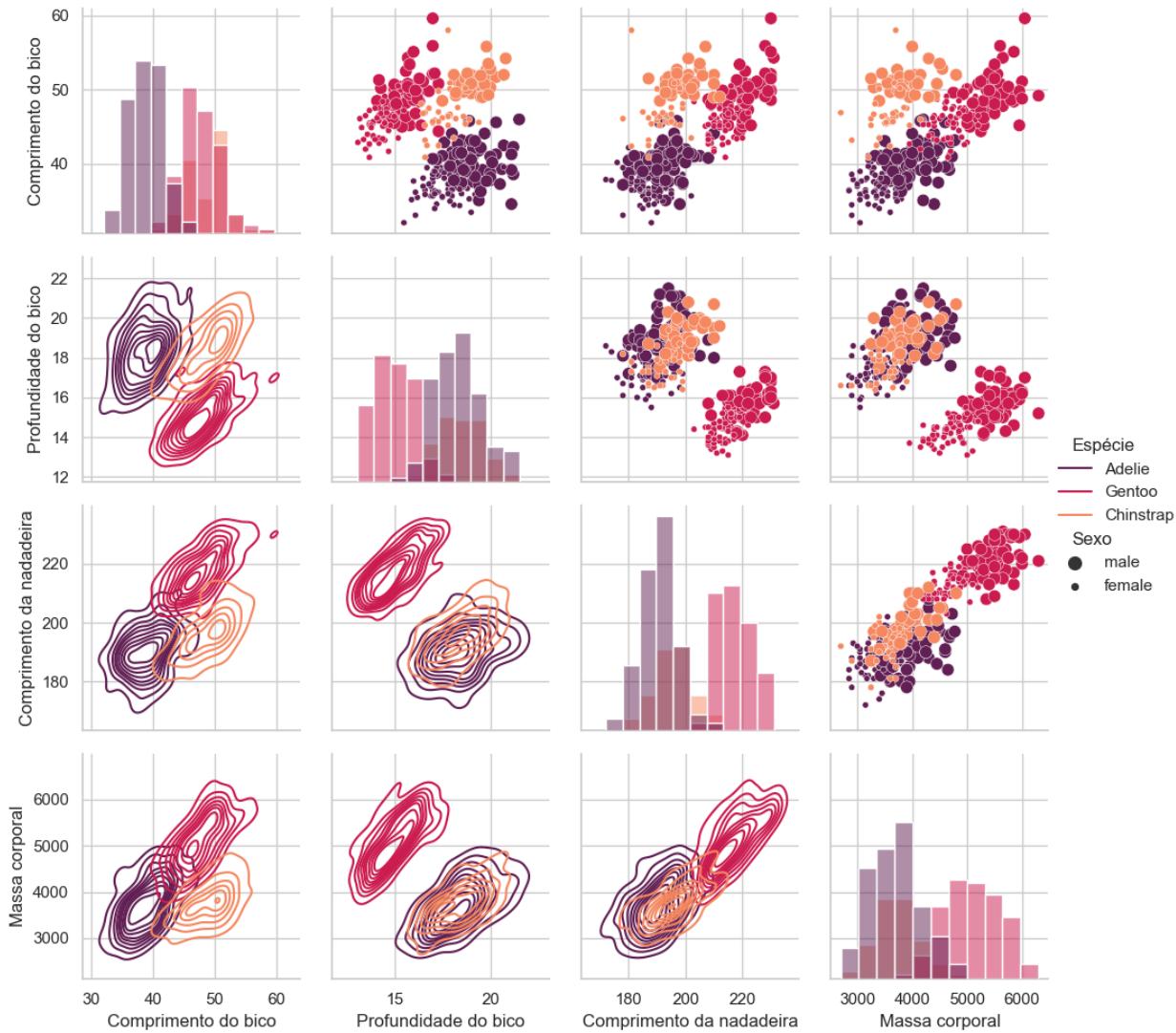


Faça as representações destes tipos de gráficos com os dados do conjunto Iris.

Uma representação mais completa pode ser feita usando mais de um tipo de gráfico. Na parte inferior da diagonal, podemos definir um tipo, na diagonal um segundo tipo e o terceiro tipo de gráfico fica definido para aparecer na parte superior da matriz. Além disso, podemos usar outra variável para representar o tamanho de cada ponto representado no gráfico de pontos.

```
g = sns.PairGrid(data = penguins, hue = 'Espécie', palette = 'rocket')
g.map_diag(sns.histplot)
g.map_upper(sns.scatterplot, size = penguins['Sexo'])
g.map_lower(sns.kdeplot)
g.add_legend(title = '', adjust_subtitles = True)
```

Note que a variável escolhida para ajustar os tamanhos de pontos foi convertida para binária de forma automática, pois seus valores são nominais.



Outra maneira de representar os dados de classificação pode ser feita usando 3 atributos representados em gráficos 3D. Utilizando a biblioteca `plotly`, podemos indicar os atributos escolhidos das coordenadas x, y e z e escolher a separação de classes por meio do atributo de cor. Neste primeiro exemplo, utilizamos o atributo de espécie para separar os dados:

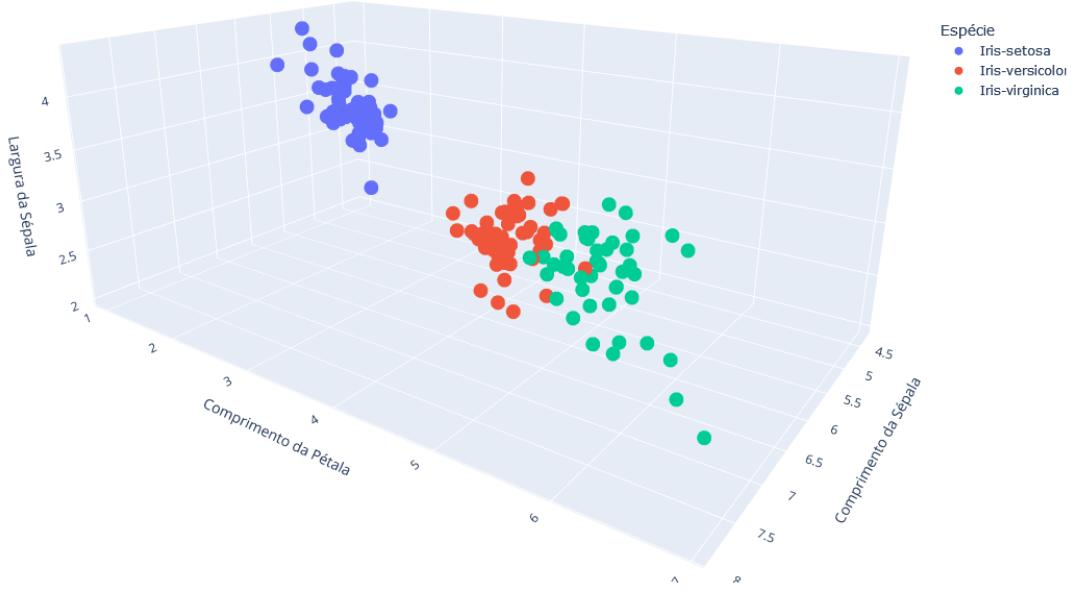
```
import pandas as pd
import plotly.io as pio
import plotly.express as px

iris = pd.read_csv('C:/dados/iris.csv')

pio.renderers
pio.renderers.default = 'browser'

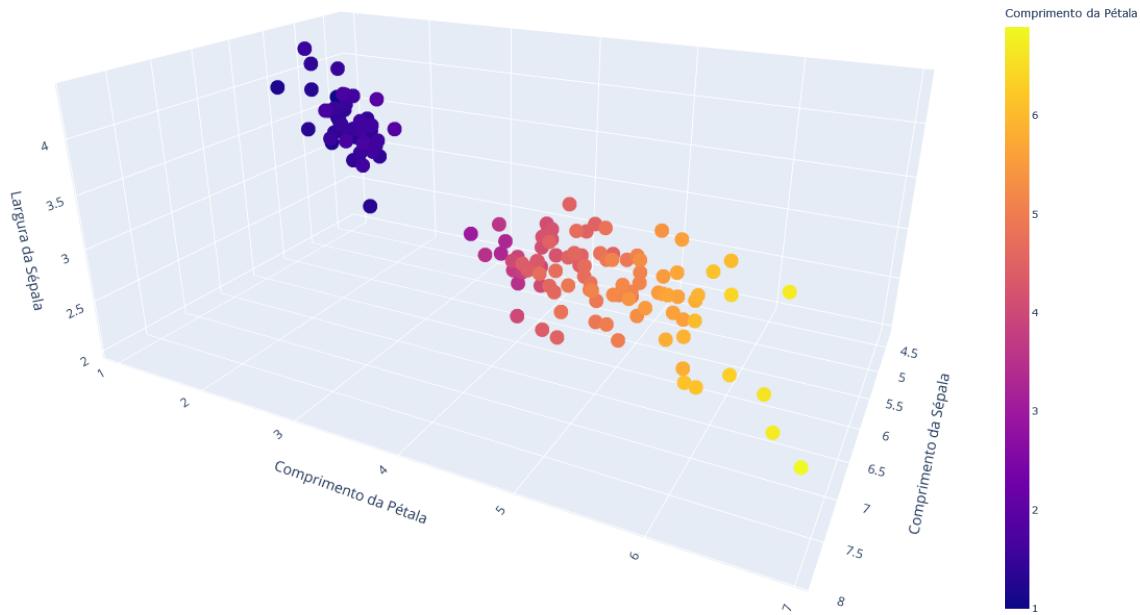
fig = px.scatter_3d(iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
                    z = 'Largura da Sépala', color = 'Espécie')

fig.show()
```



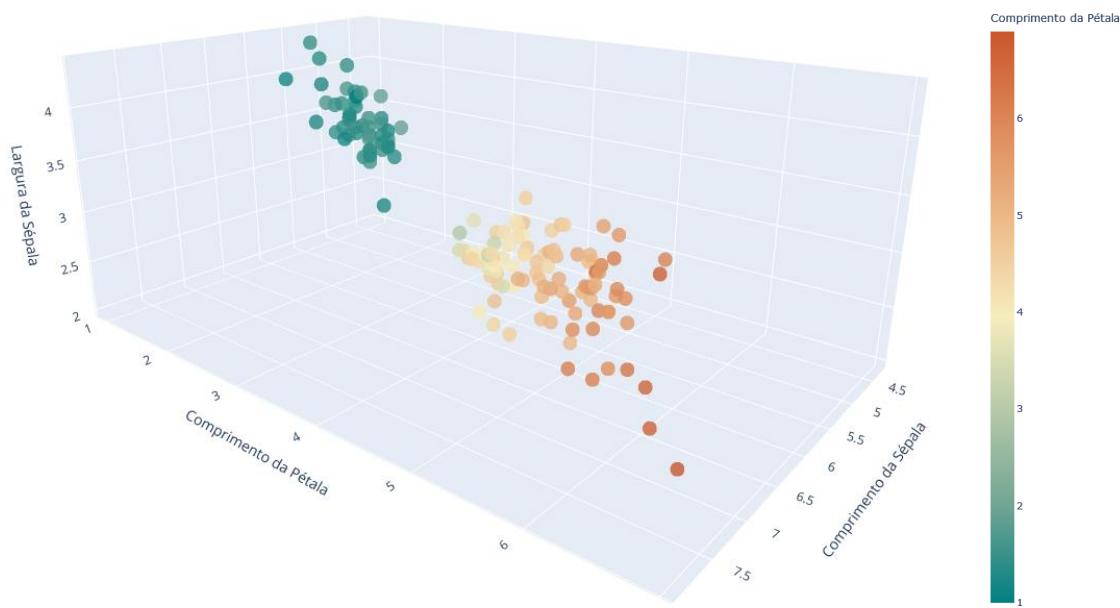
Se escolhermos outro atributo, temos a representação com a variação de valores da variável escolhida. Neste exemplo, temos a variação de brilho encontrada da visualização dos dados com o critério de separação da variável de comprimento da pétala.

```
fig = px.scatter_3d(iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
                     z = 'Largura da Sépala', color = 'Comprimento da Pétala')
```



Podemos definir o atributo de opacidade e escolher a escala de cores. No exemplo anterior, podemos usar o atributo `color_continuous_scale` para definir a variação de cores do gráfico. Com a propriedade `geyser` e opacidade 80%, temos a seguinte representação com a variável visual de brilho nos dados:

```
fig = px.scatter_3d(iris, x = 'Comprimento da Sépala', y = 'Comprimento da Pétala',
                     z = 'Largura da Sépala', color = 'Comprimento da Pétala', opacity = 0.8,
                     color_continuous_scale = 'geyser')
```

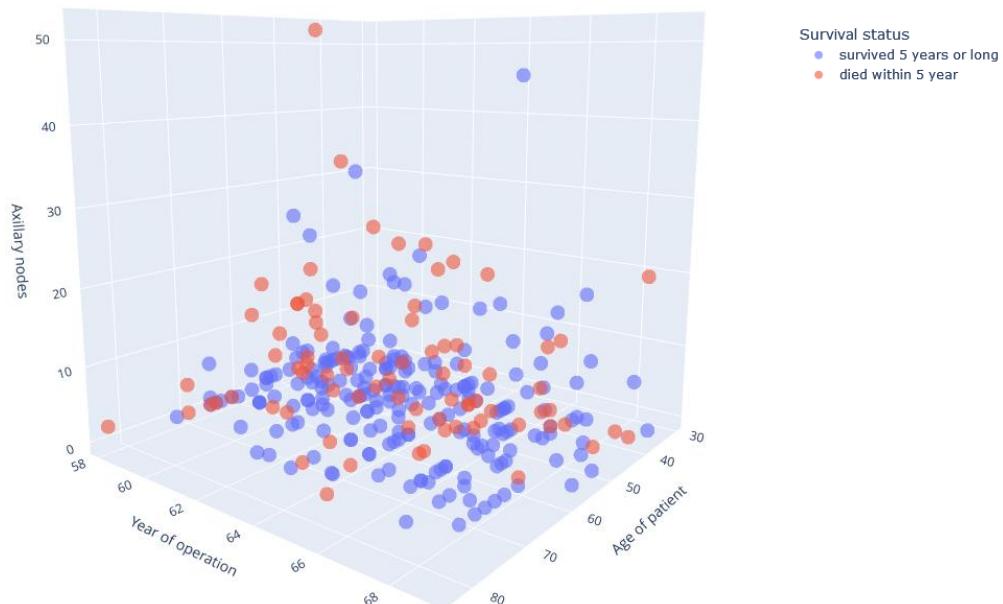


Os nomes dos mapas de cores que podemos usar na biblioteca **plotly**, de acordo com os valores das variáveis contínuas dos dados, são os seguintes:

```
{aggrnyl, agsunset, algae, amp, armyrose, balance, blackbody, bluered, blues, blugrn, bluyl, brbbg, brwnyl, bugn, bupu, burg, burgyl, cividis, curl, darkmint, deep, delta, dense, earth, edge, electric, emrld, fall, geyser, gnbu, gray, greens, greys, haline, hot, hsv, ice, icefire, inferno, jet, magenta, magma, matter, mint, mrybm, mygbm, oranges, orrd, oryel, oxy, peach, phase, picnic, pinkyl, piyg, plasma, plotly3, portland, prgn, pubu, pubugn, puor, purd, purp, purples, purpor, rainbow, rdbu, rdgy, rdpu, rdylbu, rdylgn, redor, reds, solar, spectral, speed, sunset, sunsetdark, teal, tealgrn, tealrose, tempo, temps, thermal, tropic, turbid, turbo, twilight, viridis, ylgn, ylgnbu, ylorbr, ylorrd}
```

No exemplo de dados de classificação **haberman**, que contém casos de um estudo realizado entre 1958 e 1970 em um hospital de Chicago sobre a sobrevivência de pacientes que foram submetidos a cirurgias de câncer da mama, temos os dados representados com opacidade de 60%.

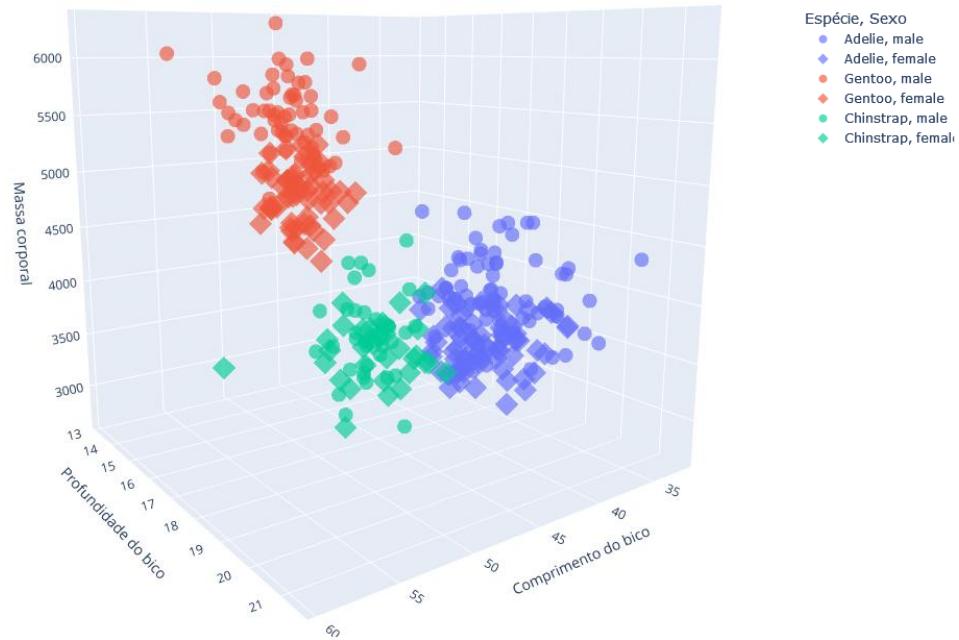
```
fig = px.scatter_3d(haberman, x = 'Age of patient', y = 'Year of operation',  
z = 'Axillary nodes', color = 'Survival status', opacity = 0.6)
```



Dados disponíveis em: <https://archive.ics.uci.edu/ml/datasets/haberman%27s+survival>

Para deixar os dados representados com mais detalhes, o atributo `symbol` pode ser usado para separar os dados com mais uma variável. No exemplo dos pinguins, podemos usar este atributo para separar o sexo dos animais.

```
fig = px.scatter_3d(penguin, x = 'Comprimento do bico', y = 'Profundidade do bico',
                     z = 'Massa corporal', color = 'Espécie', symbol = 'Sexo', opacity = 0.65)
```



Defina outras combinações de atributos dos exemplos de dados Iris e Pinguins para criar novas representações gráficas destes conjuntos. Utilize outro conjunto de dados diferente para fazer a análise de representação gráfica com pelo menos 3 atributos em gráficos 3D.

O **Movimento** é a última variável visual, que pode ser usada com qualquer atributo de um conjunto de dados, desde que exista uma indicação com a informação sobre as mudanças de valores. Uma aplicação comum deste tipo de associação de atributo com movimento mostra as mudanças de posicionamento de marcadores ou determina uma trajetória variando as cores ou opacidade. A direção precisa ser considerada para indicar o sentido do movimento.

O primeiro exemplo com a associação de movimento nos dados mostra com a variação de opacidade as trajetórias dos pontos de duas hélices cilíndricas variando-se as medidas das coordenadas dos respectivos pontos.

```
import matplotlib.pyplot as plt
import numpy as np

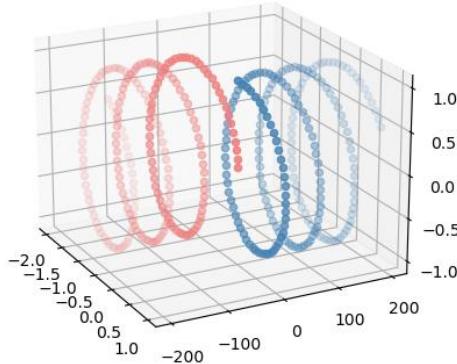
ax = plt.figure().add_subplot(projection = '3d')

d = 1
e = 10
op = 0.9

z = np.arange(0, 200, 1)
x = d * np.sin(z/e)
y = d * np.cos(z/e)

for k in z:
    op*=0.99
    ax.scatter(x[k], z[k], y[k], zdir = 'z', color = 'steelblue', alpha = op)
    ax.scatter(y[k] - d, d - z[k], x[k], zdir = 'z', color = 'lightcoral', alpha = op)

plt.show()
```

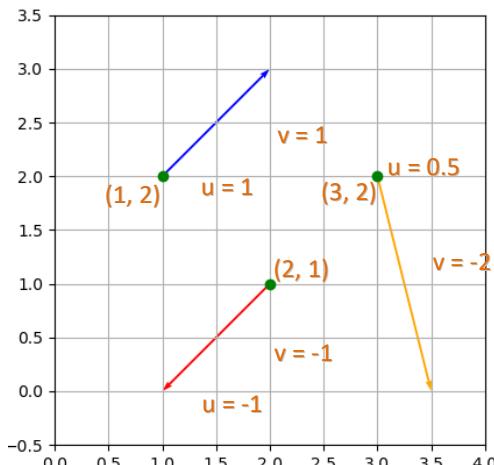


3.2. Dados vetoriais

No Cálculo Vetorial e na Física, um campo vetorial é uma atribuição de um vetor para cada ponto de um subconjunto do espaço. Um campo vetorial no plano pode ser visualizado como uma coleção de setas, cada uma com sua respectiva magnitude e direção. Os campos vetoriais são usados para modelar fenômenos climáticos, a velocidade e direção de um fluido em movimento no espaço, ou a força e direção de uma força magnética ou gravitacional conforme ela muda de um ponto para outro. A combinação de escalares e vetores define uma estrutura conhecida como Tensor.

A função `quiver` da biblioteca `matplotlib` pode ser usada para representações gráficas de campos vetoriais. Considere o exemplo com 3 vetores. Para campos vetoriais 2D, precisamos definir 4 vetores: X e Y, que são as coordenadas das origens dos vetores, U e V, que definem os respectivos pontos finais com as setas dos vetores.

Os valores de cada componente U e V são coordenadas relativas de X e Y, respectivamente, que definem a direção e a magnitude de cada vetor. Podemos definir um vetor de cores para cada representação de vetor e a escala, que pode ser usada para modificar proporcionalmente os tamanhos dos vetores.



```

import matplotlib.pyplot as plt

X = [1, 2, 3]
Y = [2, 1, 2]
U = [1, -1, 0.5]
V = [1, -1, -2]
cor = ['blue','red','orange']

fig, ax = plt.subplots()
Q = ax.quiver(X, Y, U, V, color = cor, units = 'xy', width = 0.02, scale = 1)

ax.plot(X, Y, 'og')
ax.set_aspect('equal', 'box')
plt.xlim([0, 4])
plt.ylim([-0.5, 3.5])
plt.grid()

```

```
plt.show()
```

Com os dados de um arquivo csv, podemos representar o campo vetorial 2D com o código da biblioteca matplotlib mostrado a seguir. Como se trata de um conjunto com muitos vetores, podemos usar uma função que atribui as cores de acordo com as medidas dos vetores relativos U e V. Neste caso, utilizamos a função hypot, que faz o cálculo da medida de hipotenusa de um triângulo retângulo: `sqrt(U**2 + V**2)`.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

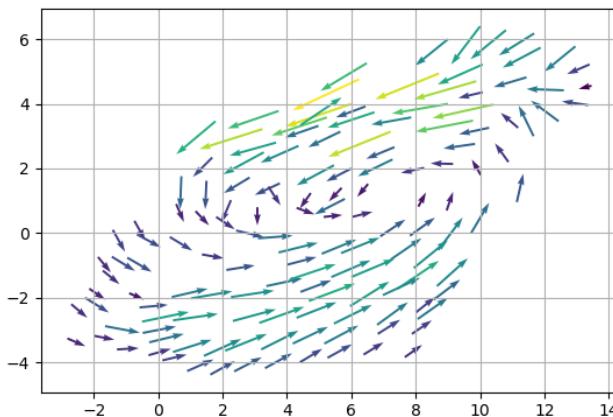
dados = pd.read_csv('C:/dados/dados_teste.csv')

X = dados.loc[:, 'x']
Y = dados.loc[:, 'y']
U = dados.loc[:, 'u']
V = dados.loc[:, 'v']
fig, ax = plt.subplots()

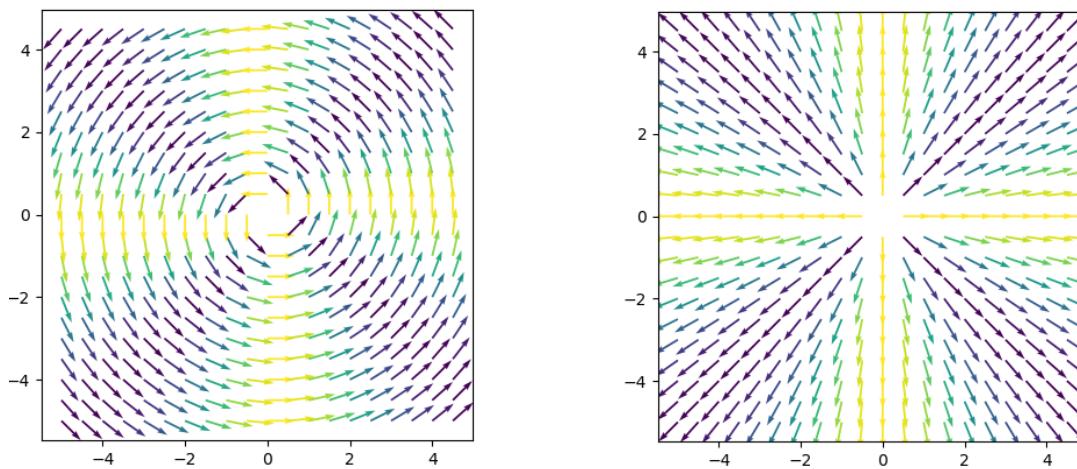
M = np.hypot(U, V)
Q = ax.quiver(X, Y, U, V, M, units = 'x', width = 0.07, scale = 1.2)

ax.set_aspect('equal', 'box')

plt.grid()
plt.show()
```



Se definirmos uma grade para as variáveis X e Y, podemos definir funções para os valores dos vetores U e V do campo vetorial. Utilizamos a função `meshgrid` para criar a grade no intervalo [-5, 5] com espaçamento 0.5 entre cada variável. Estes espaçamentos foram feitos com a função `arange`.



```
X, Y = np.meshgrid(np.arange(-5,5,0.5),np.arange(-5,5,0.5))
```

```

U = -Y/np.hypot(X, Y)
V = X/np.hypot(X, Y)

M = np.hypot(U**3, V**3)
fig, ax = plt.subplots()
Q = ax.quiver(X, Y, U, V, M, units = 'xy', width = 0.05, scale = 1.5)

```

O segundo gráfico tem as seguintes funções para U e V:

```

U = X/np.hypot(X, Y)
V = Y/np.hypot(X, Y)

```

Nas representações de campos vetoriais 3D, temos que definir as coordenadas dos vetores Z e W. Além disso, temos a opção de normalizar os vetores e os atributos `lw`, que define a espessura das linhas, e `length`, que tem a mesma função de escala dos vetores 2D. No exemplo mostrado a seguir, usamos as linhas com espessura 0.8 e escala 0.4.

```

import matplotlib.pyplot as plt

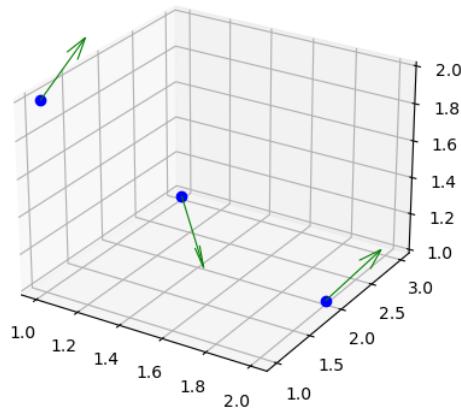
ax = plt.figure().add_subplot(projection = '3d')

x = [1, 2, 1]
y = [1, 2, 3]
z = [2, 1, 1]
u = [1, 2, 1]
v = [1, 1, -1]
w = [2, 3, -2]

Q = ax.quiver(x, y, z, u, v, w, length = 0.4, normalize = True, color = 'green', lw = 0.8)

ax.plot(x, y, z, 'ob')
plt.show()

```



Nos campos vetoriais 3D, podemos definir uma grade com as variáveis X, Y e Z e funções entre as coordenadas de U, V e W. A função para atribuir as cores utiliza os limites mínimo e máximo de c.

```

import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection = '3d')

x, y, z = np.meshgrid(np.arange(-2, 2, 0.5), np.arange(-2, 2, 0.5), np.arange(-2, 2, 0.5))
u = x - z
v = y + z
w = z + 1

cor = np.arcsinh(v, u)

```

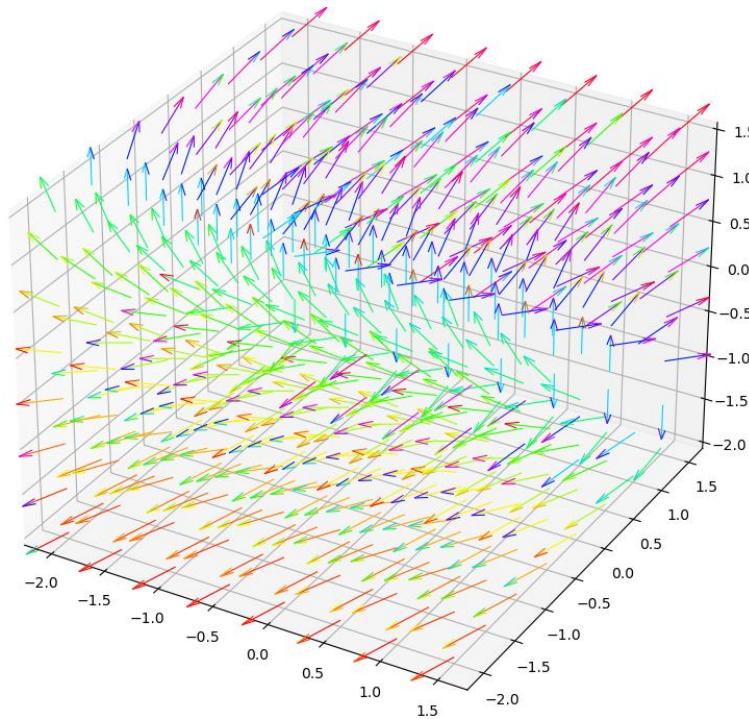
```

cor = (cor.flatten() - cor.min()) / cor.ptp()
cor = plt.cm.hsv(cor)

a = ax.quiver(x, y, z, u, v, w, length = 0.4, normalize = True, colors = cor, lw = 0.8)

plt.show()

```



Os dados da simulação do vórtice de Burger podem ser representados graficamente com o código mostrado a seguir. Este campo vetorial apresenta o movimento do fluido induzido pela co-rotação de dois discos, enquanto retira o fluido axialmente por meio de eixos de acionamento ocos. Este vórtice simula a criação de redemoinhos turbulentos em escala dissipativa nas zonas costeiras e próximas à superfície.

O banco de dados desta simulação possui 1.794.621 vetores, e vamos fazer a representação de apenas uma parte deste conjunto. Vamos criar um conjunto de índices que lê os dados em intervalos de 750 em 750 com a função arange. A biblioteca `scipy.io` foi usada para leitura dos dados em formato .mat.

```

import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat

ax = plt.figure().add_subplot(projection = '3d')
data_dict = loadmat('c:/dados/flow_field.mat')

x = np.transpose(data_dict['X'])
y = np.transpose(data_dict['Y'])
z = np.transpose(data_dict['Z'])
u = np.transpose(data_dict['V_x'])
v = np.transpose(data_dict['V_y'])
w = np.transpose(data_dict['V_z'])

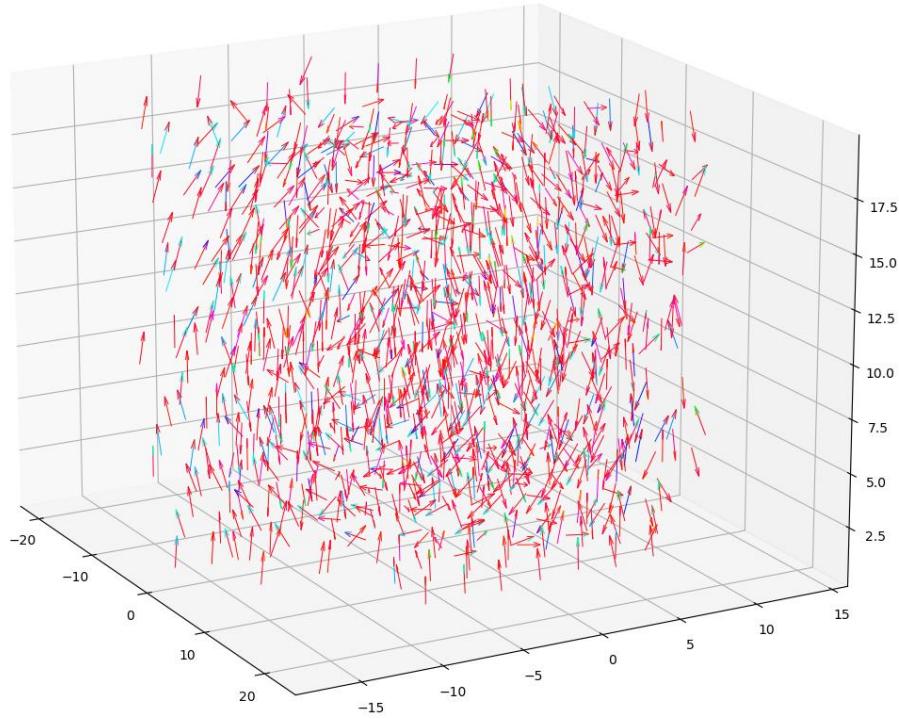
ind = np.arange(0, len(x), 1500)

c = (np.arctan2(u, v))
c = (c.flatten() - c.min()) / c.ptp()
c = np.concatenate((c, np.repeat(c, 2)))
c = plt.cm.hsv(c)

ax.quiver(x[ind], y[ind], z[ind], u[ind], v[ind], w[ind], colors = c, length = 1.5,
           normalize = True, lw = 0.7)

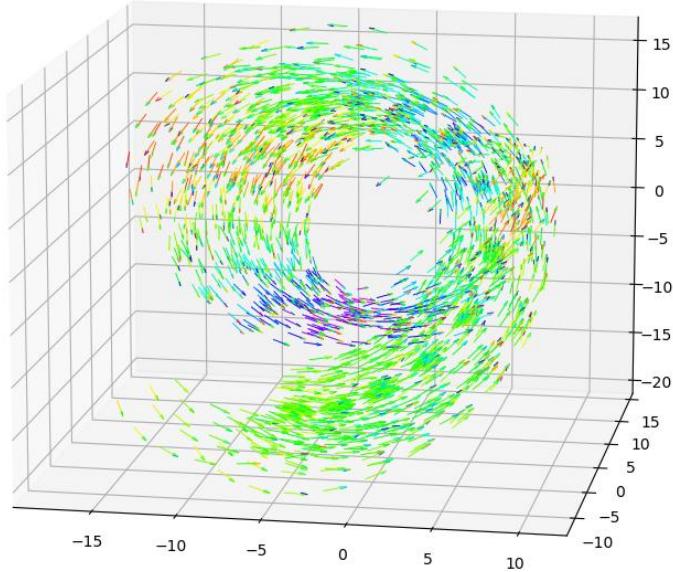
```

```
plt.show()
```



Dados disponíveis em: <https://www.bco-dmo.org/dataset/834530>

Na imagem mostrada a seguir, temos outro exemplo de um conjunto vortex representado de forma análoga.



Dados disponíveis em: <https://raw.githubusercontent.com/plotly/datasets/master/vortex.csv>

O modelo simulado de um tornado foi criado por Roger Crawfis, e está definido em um grid com espaçamento de 128 entradas nos intervalos [-10, 10] para as coordenadas X, Y e Z. O código mostrado a seguir usa a mesma ideia do exemplo anterior para representar os dados deste campo vetorial. Utilizamos a biblioteca **netCDF4** para fazer a leitura do arquivo com extensão .nc.

```
import numpy as np
import matplotlib.pyplot as plt
import netCDF4 as nc

data = nc.Dataset('C:/dados/tornado3d.nc')
ax = plt.figure().add_subplot(projection = '3d')
```

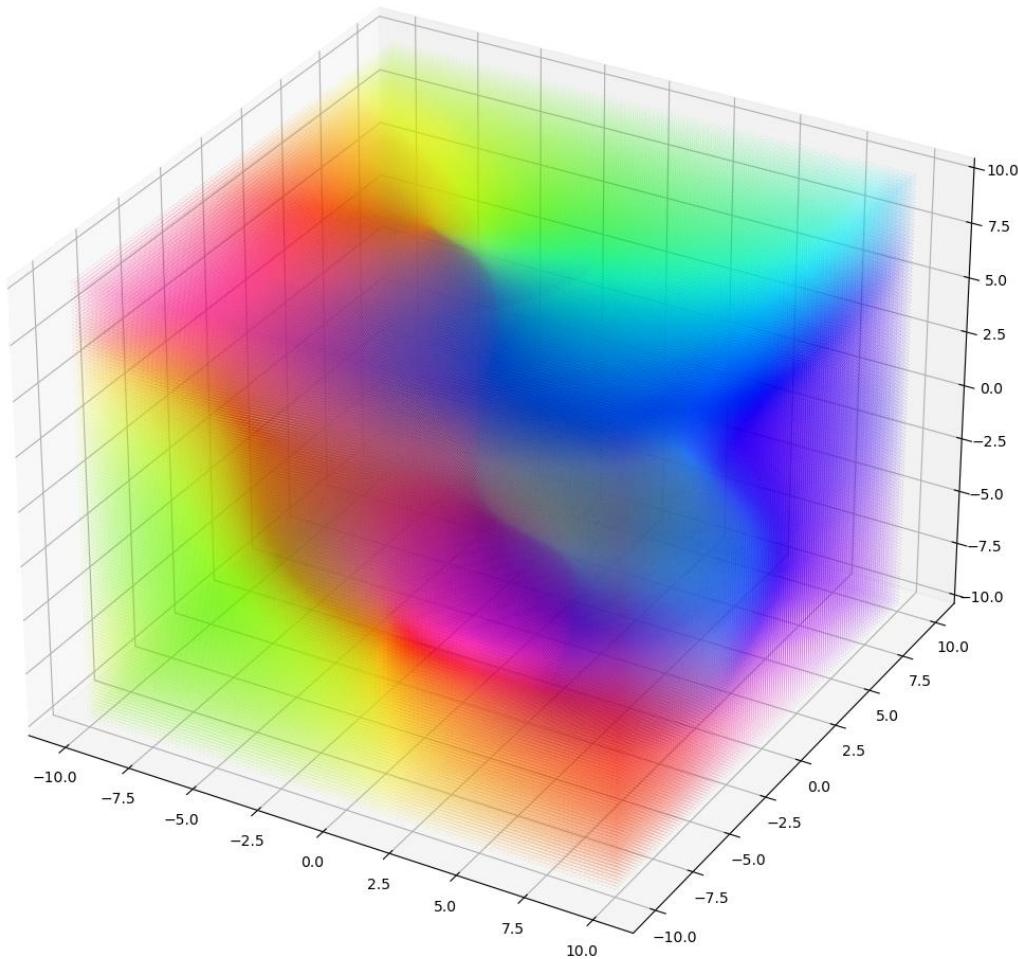
```
u = np.array(data['u'])
v = np.array(data['v'])
w = np.array(data['w'])
xx = np.array(data['xdim'])
yy = np.array(data['ydim'])
zz = np.array(data['zdim'])

x, y, z = np.meshgrid(xx, yy, zz)

c = (np.arctan2(v, u))
c = (c.flatten() - c.min()) / c.ptp()
c = np.concatenate((c, np.repeat(c, 2)))
c = plt.cm.hsv(c)

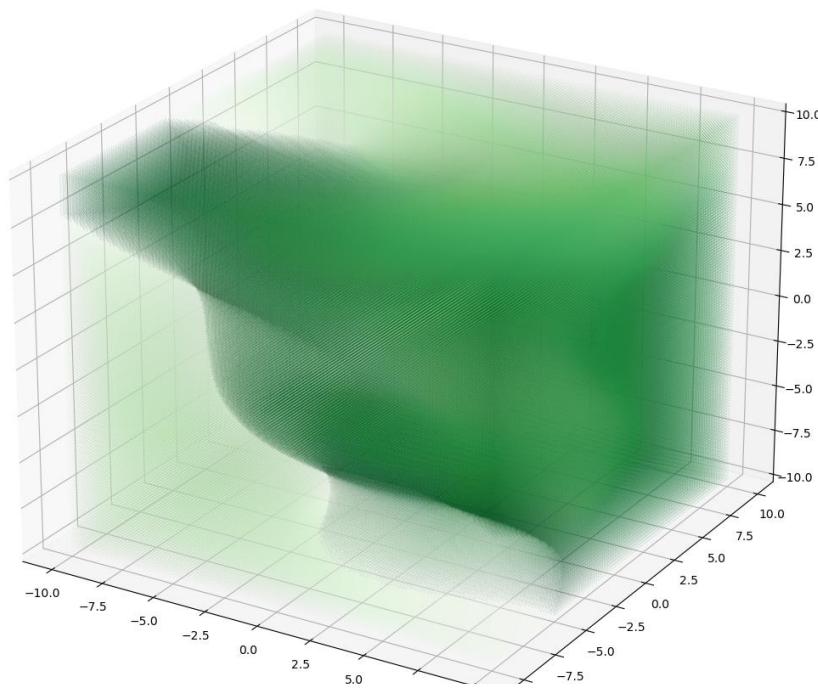
a = ax.quiver(x, z, y, w, u, v, colors = c, length = 0.05, normalize = True, lw = 2,
               alpha = 0.1)

plt.show()
```



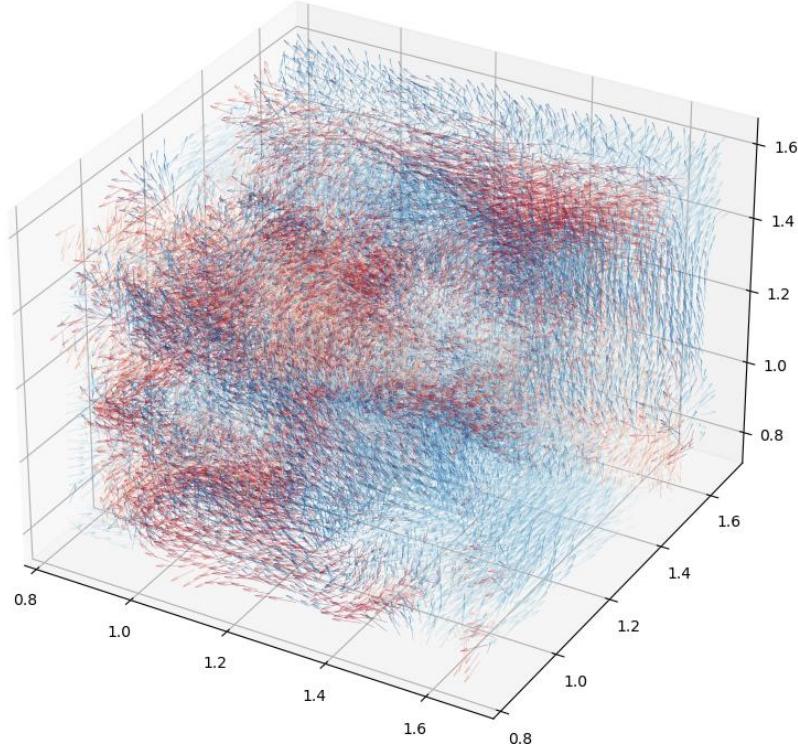
Mudando a configuração do mapa de cores, temos a seguinte representação para os dados do conjunto `tornado3d`:

```
c = plt.cm.Greens(c)
```



Dados disponíveis em: <https://cgl.ethz.ch/research/visualization/data.php>

O conjunto de dados Cloud-topped Boundary Layer contém uma simulação de camada limite de nuvem que foi simulada com o modelo UCLA-LES. Temos a representação de várias nuvens cumulus crescentes neste conjunto de dados cedido pelo Centro Alemão de Computação Climática (DKRZ) e pelo Instituto Max Planck de Meteorologia (MPI-M). A representação gráfica mostrada a seguir contém 10% do conjunto de dados, usando o mesmo código de representação do tornado3d.



Dados disponíveis em: <https://cgl.ethz.ch/research/visualization/data.php>

Um gráfico **Streamline** (linhas de fluxo) é uma representação baseada em um campo vetorial 2D interpretado como um campo de velocidade. O streamline é composto de curvas fechadas tangentes ao campo de velocidade. No caso de um campo de velocidade estacionário, as linhas de corrente coincidem com as trajetórias. Em figuras aerodinâmicas, precisamos das informações sobre os intervalos uniformemente espaçados de valores x e y.

Assim como fizemos nos campos vetoriais, criamos a grade meshgrid com as coordenadas x e y, e os valores de velocidade são interpolados por meio das chamadas linhas de corrente u e v. As linhas de fluxo são determinadas no domínio das variáveis x e y.

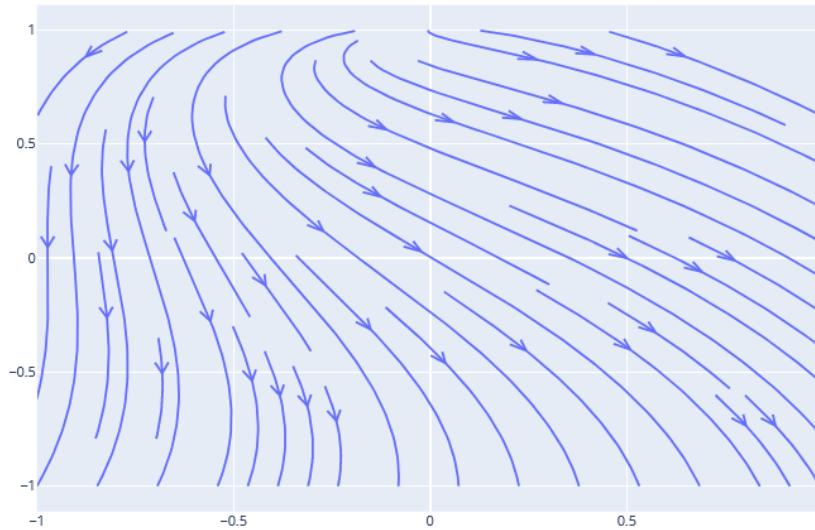
Nas streamlines podemos visualizar as direções do fluxo que libera partículas e calculamos uma série de posições de partículas com base no campo vetorial. O código mostrado a seguir contém a configuração da grade x e y e das funções que determinam as linhas de corrente u e v.

```
import plotly.figure_factory as ff
import numpy as np

x = np.linspace(-1, 1, 10)
y = np.linspace(-1, 1, 10)
Y, X = np.meshgrid(x, y)
u = 1 - X**2 + Y
v = -1 + X - Y**2

fig = ff.create_streamline(x, y, u, v, arrow_scale = 0.05)

fig.show()
```



Em gráficos Streamtube (tubo de fluxo), os atributos incluem o domínio com as variáveis x, y e z em uma grade 3D do campo vetorial, e as componentes do campo vetorial u, v e w, das variáveis x, y e z. Além disso, você pode usar o atributo **start** para determinar a posição inicial do Streamtube. O código mostrado a seguir contém os dados da representação do fluxo de vento com as funções da biblioteca plotly.

```
import plotly.graph_objects as go
import pandas as pd
import plotly.io as pio

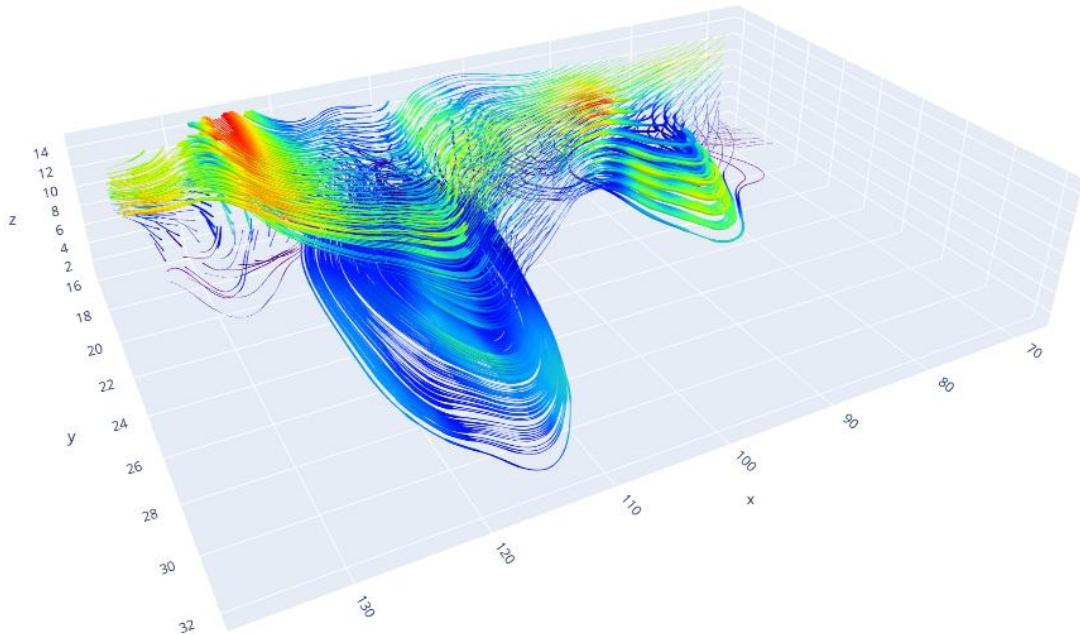
pio.renderers
pio.renderers.default = 'browser'

df = pd.read_csv('C:/dados/streamtube-wind.csv')

fig = go.Figure(data = go.Streamtube(x = df['x'], y = df['y'], z = df['z'], u = df['u'],
                                         v = df['v'], w = df['w'], sizeref = 0.3, colorscale = 'rainbow', maxdisplayed = 3000))

fig.update_layout(scene = dict(aspectratio = dict(x = 1.5, y = 1, z = 0.3)))

fig.show()
```



Dados disponíveis em: <https://github.com/plotly/datasets>

3.3. Tratamentos dos dados

Em alguns casos, os dados brutos (raw data) devem ser considerados na visualização. Na maioria das aplicações médicas, os dados não sofrem modificações para serem visualizados, pois algumas informações importantes podem ser perdidas ou valores adicionais poderiam aparecer nos modelos modificados.

De acordo com o tipo de dado ou com a técnica de visualização usada, os dados necessitam de um tratamento de pré-processamento, para verificar dados faltantes, outliers ou erros. Podemos citar alguns métodos para executar o pré-processamento de dados: Estatística, Interpolação de valores faltantes, Limpeza dos dados, Normalização, Segmentação, Amostragem e Redução de Dimensionalidade.

A aplicação de **Métodos Estatísticos** é muito importante para analisar agrupamentos de dados, eliminar variáveis redundantes (análise de correlação), além de detectar erros ou outliers nos dados.

Nos conjuntos de dados reais possuem, geralmente, dados faltantes ou com erros. Estes dados podem aparecer pelo mau funcionamento de um sensor, por uma entrada em branco de uma pesquisa ou pela omissão de algum dado. Quando o valor de um atributo possui erro, pode ter relação com uma falha humana e dificilmente conseguimos detectá-lo.

Algumas estratégias para corrigir estes problemas são:

- **descartar a instância com erros:** trata-se de uma medida drástica, mas é comum, e devemos ter o cuidado de analisar se a qualidade das instâncias restantes continua significante para a análise dos dados. Este descarte pode causar uma grande perda de informação, principalmente quando o conjunto de dados tem muitos dados faltantes ou com erros;
- **associar um valor sentinelas:** este valor indica o final de uma sequência de dados. Por exemplo, se os dados variam de -10 a 10, podemos escolher um outro valor fora deste intervalo: -15. Desta forma, quando os dados forem visualizados, as instâncias com valores faltantes ou com erros podem ser identificadas. Devemos ter cuidado de não levar em consideração estes valores sentinelas em algum processamento para alguma medição estatística;
- **associar um valor médio:** nestes casos, podemos fazer a substituição do valor faltante ou com erros por um valor médio calculado da variável em questão. Com esta substituição os dados sofrem poucas alterações sob o ponto de vista estatístico. Porém, devemos tomar cuidado com os outliers, pois se forem considerados os valores com erros deste tipo para o cálculo de valor médio, os dados ficam com mais erros; ou
- **associar um valor baseado nos vizinhos mais próximos:** trata-se da melhor abordagem para substituição de valores com erros ou faltantes. Nestes casos, definimos uma instância muito parecida com àquela em

questão, com base nos valores das outras variáveis. Quando encontrarmos a instância P, mais parecida, os valores faltantes ou errôneos são substituídos pelos valores desta instância P.

ID	Espécie	Ilha	Comprimento do bico	Profundidade do bico	Comprimento da nadadeira	Massa corporal	Sexo	Ano
1	Adelie	Torgersen	39.1	18.7	181	3750	male	2007
2	Adelie	Torgersen	39.5	17.4	186	3800	female	2007
3	Adelie	Torgersen	40.3	18	195	3250	female	2007
4	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
5	Adelie	Torgersen	36.7	19.3	193	3450	female	2007
6	Adelie	Torgersen	39.3	20.6	190	3650	male	2007
7	Adelie	Torgersen	38.9	17.8	181	3625	female	2007
8	Adelie	Torgersen	39.2	19.6	195	4675	male	2007
9	Adelie	Torgersen	34.1	→ NA	193	3475	NA	2007
10	Adelie	Torgersen	42	20.2	190	4250	NA	2007
11	Adelie	Torgersen	37.8	17.1	186	3300	NA	2007
12	Adelie	Torgersen	→ NA	17.3	180	3700	NA	2007
13	Adelie	Torgersen	41.1	17.6	182	3200	female	2007
14	Adelie	Torgersen	38.6	21.2	191	3800	male	2007
15	Adelie	Torgersen	34.6	21.1	198	4400	male	2007
16	Adelie	Torgersen	→ 36.6	17.8	185	3700	female	2007
17	Adelie	Torgersen	38.7	19	195	3450	female	2007
18	Adelie	Torgersen	42.5	20.7	197	4500	male	2007
19	Adelie	Torgersen	34.4	18.4	184	3325	female	2007
20	Adelie	Torgersen	46	21.5	194	4200	male	2007
21	Adelie	Biscoe	37.8	18.3	174	3400	female	2007
...

O processo de **Normalização** transforma um conjunto de dados para uma mesma escala, além de aplicar filtros de detecção de dados repetidos. No exemplo mostrado com os 21 primeiros dados dos pinguins, podemos notar que os dados possuem escalas bem diferentes, que podem causar alguma discrepância quando visualizadas em gráficos. Dividindo os dados de Comprimento do bico, Profundidade do bico, Comprimento da nadadeira e Massa Corporal pelos maiores valores de cada atributo (59.6, 21.5, 231 e 6300), temos os 5 primeiros dados com os seguintes valores no intervalo [0, 1]:

$$d'_i = \frac{d_i}{\max_j \{d_j\}}.$$

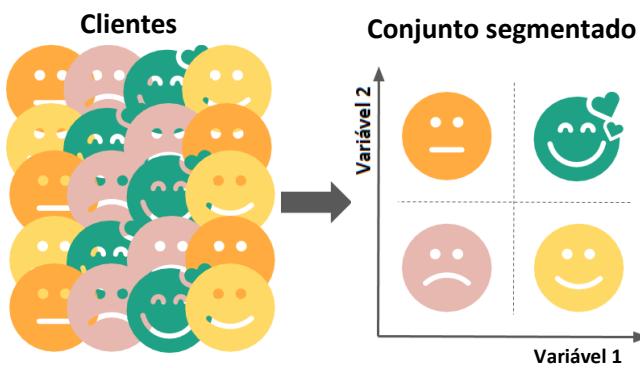
ID	Espécie	Ilha	Comprimento do bico	Profundidade do bico	Comprimento da nadadeira	Massa corporal	Sexo	Ano
1	Adelie	Torgersen	0,65604	0,86977	0,78355	0,59524	male	2007
2	Adelie	Torgersen	0,66275	0,80930	0,80519	0,60317	female	2007
3	Adelie	Torgersen	0,67617	0,83721	0,84416	0,51587	female	2007
4	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
5	Adelie	Torgersen	0,61577	0,89767	0,83550	0,54762	female	2007
...

Outra maneira de normalizar os dados pode ser feita utilizando o valor mínimo de cada atributo corresponder ao número 0 e o valor máximo de cada atributo corresponder ao número 1. Usando os valores máximos e mínimos (32.1, 13.1, 172 e 2700) dos atributos, temos os 5 primeiros dados no intervalo [0, 1]:

$$d'_i = \frac{d_i - \min_j\{d_j\}}{\max_j\{d_j\} - \min_j\{d_j\}}.$$

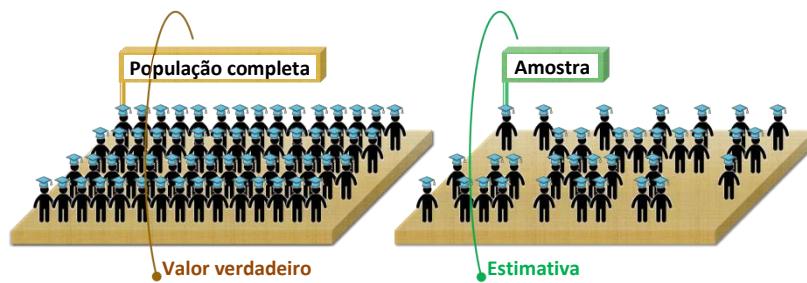
ID	Espécie	Ilha	Comprimento do bico	Profundidade do bico	Comprimento da nadadeira	Massa corporal	Sexo	Ano
1	Adelie	Torgersen	0,25455	0,66667	0,15254	0,29167	male	2007
2	Adelie	Torgersen	0,26909	0,51190	0,23729	0,30556	female	2007
3	Adelie	Torgersen	0,29818	0,58333	0,38983	0,15278	female	2007
4	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
5	Adelie	Torgersen	0,16727	0,73810	0,35593	0,20833	female	2007
...

Muitas vezes, precisamos fazer uma **Segmentação** dos dados, ou seja, separá-los por meio de uma ou mais características para análise ou normalização. Por exemplo, podemos aplicar as normalizações de dados dos pinguins separando os dados pelo atributo de Sexo e Ano.



Fonte: <https://github.com/rppradhan08/rfm-segmentation>

Em alguns conjuntos de dados podemos fazer a **Amostragem**. Trata-se da técnica estatística comum usada, por exemplo, em pesquisas de eleições ou de opinião. Se um pesquisador tem um conjunto de 6 milhões de dados sobre as características comuns de 10 tipos de doenças de pele, não precisa analisar o conjunto todo para estabelecer um diagnóstico. Basta selecionar um grupo de dados representativo dentre os 6 milhões disponíveis, esperando-se que seja suficiente para tornar os resultados precisos.



Fonte: <https://stats.mom.gov.sg/SL/Pages/How-Reliable-Is-A-Sample-Concepts-and-Definitions.aspx>

As amostras podem ser criadas com diferentes formas e tamanhos, pois existem vários métodos de amostragem de dados. Podemos dividí-los em amostragens probabilísticas e não probabilísticas.

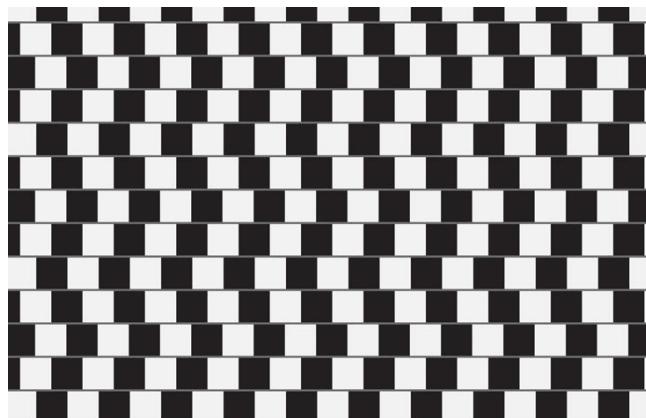
Em uma amostragem probabilística, as amostras de uma população são escolhidas usando um método baseado em métodos estatísticos. Podemos selecionar números aleatórios que correspondem aos identificadores no conjunto de dados e garantem que todos os dados têm a mesma chance de seleção. Este método oferece a melhor chance de criar uma amostra verdadeiramente representativa da população.

Por outro lado, a amostragem não probabilística cria uma amostra de dados escolhidos com base no julgamento subjetivo de um analista. Isso significa que nem todos os pontos da população têm chance de serem selecionados. Este método oferece uma chance menor de criar uma amostra que ilustre com precisão os dados da população.

Em alguns casos, podemos mapear variáveis nominais para números, transformando os domínios dos valores destas variáveis. No conjunto de dados dos pinguins, podemos usar as variáveis 0 – male e 1 – female para o atributo de Sexo. Os dados das ilhas também podem ser transformados em: 0 – Torgersen, 1 – Biscoe e 2 – Dream.

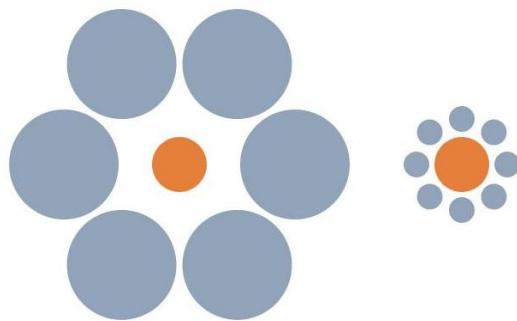
Quando temos os dados representados graficamente, precisamos ter a garantia de que os dados serão compreendidos pelos usuários. De maneira geral, definimos a percepção como o processo de reconhecer, organizar e interpretar informações sensoriais. Neste processo interpretamos o mundo ao nosso redor por meio de representações mentais do ambiente. A percepção trabalha com os sentidos humanos que geram sinais a partir do ambiente.

Algumas representações visuais de objetos podem ser mal interpretadas por dois motivos: porque elas não correspondem com o sistema perceptual da pessoa ou elas foram preparadas propositalmente para isso. Na representação de Zöllner, as linhas horizontais parecem oblíquas, mas na verdade são paralelas. A explicação para esta ilusão de ótica é que os ângulos entre as linhas curtas e as mais longas criam uma impressão de profundidade, parecendo que uma linha fica mais próxima da outra.



Disponível em: <https://www.oficinadanet.com.br/post/18957-5-ilusoes-de-otica-que-vao-mexer-com-sua-cabeca>

Na figura mostrada a seguir, qual dos círculos laranjas é maior?

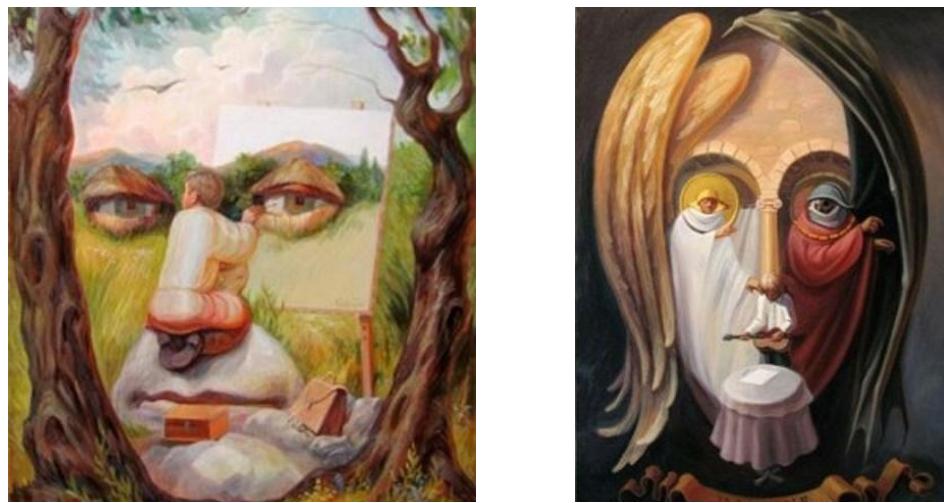


Disponível em: <https://www.oficinadanet.com.br/post/18957-5-ilusoes-de-otica-que-vao-mexer-com-sua-cabeca>

Em algumas imagens, a ambiguidade presente pode ser facilmente percebida, mas em alguns casos não conseguimos percebê-la. Em uma primeira visualização, a imagem apresenta um objeto principal, que é percebido mais facilmente do que os objetos secundários. A percepção e interpretação de objetos secundários demandam mais tempo e esforço.

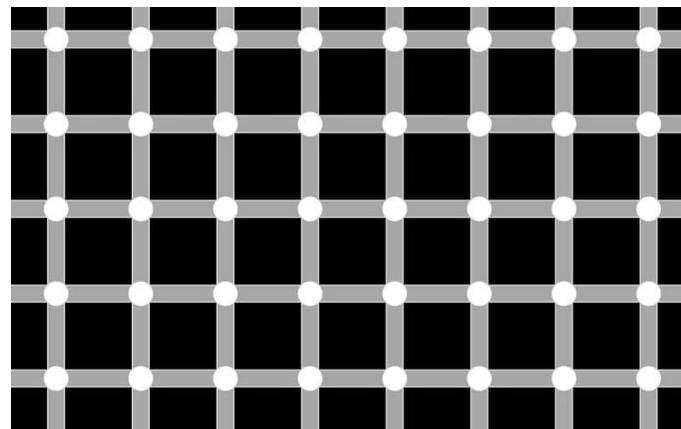
A interpretação do que enxergamos do mundo exterior é uma tarefa complexa. Existem mais de 30 áreas diferentes do nosso cérebro que são responsáveis pelo processamento da visão: algumas trabalham com o movimento, outras áreas lidam com a cor, outras com a profundidade e outras áreas são responsáveis pela interpretação de contornos.

O sistema visual do nosso cérebro simplifica as coisas, permitindo uma apreensão rápida e imperfeita da realidade, criando as ilusões de ótica.



Disponível em: <http://www.historiadasartes.com/sala-dos-professores/ilusao-de-otica/>

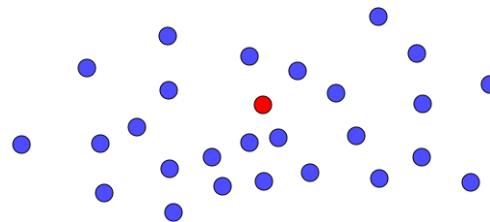
A figura mostrada a seguir revela que nosso sistema não é estático e não está sob nosso controle total. A grade de Ludimar Hermann, mostra que temos a chamada “inibição lateral” sobre a maneira que nosso cérebro responde à visualização dos círculos brancos.



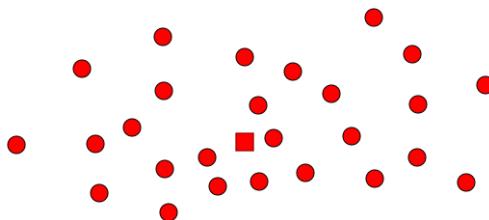
Disponível em: <https://www.oficinadanet.com.br/post/18957-5-ilusoes-de-otica-que-vao-mexer-com-sua-cabeca>

As representações para a visualização de dados devem ser construídas de maneira não ambígua e sem criar ilusões de ótica. O processo de percepção pode ser controlado (atentivo) e não-controlado (pré-atentivo). O processo pré-atentivo é rápido, paralelo e inconsciente. Porém, o processo atentivo é mais lento, sequencial e consciente e recorre à memória de curto prazo. O processo atentivo é responsável pela identificação dos objetos a partir dos estímulos visuais iniciais.

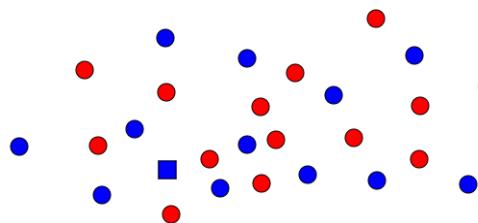
Quanto tempo precisamos para identificar a posição do círculo vermelho no conjunto mostrado a seguir?



O objeto alvo tem a propriedade visual "vermelho" e os objetos distratores não têm. Mas a cor não é a única característica visual que é pré-atentiva. A forma também pode ser utilizada como uma tarefa pré-atentiva. Existe algum quadrado vermelho (alvo) dentre vários quadrados círculos vermelhos (distratores)?



O processo pré-atentivo pode ser aproveitado na visualização para chamar a atenção do usuário para determinadas partes importantes da visualização, pois a procura de uma característica pré-atentiva permite focar rapidamente em um objeto alvo. Porém, a combinação de mais de uma característica não permite uma detecção pré-atentiva. Por exemplo, identificar se o quadrado azul (alvo) está presente entre objetos distratores que possuem essas mesmas características (cor e forma).



A combinação de características deve ser utilizada com cuidado, para não mascarar a informação que está sendo transmitida. Outras características visuais que também são consideradas pré-atentivas são: comprimento, largura, profundidade, tamanho, número, setas, interseções, proximidade, intensidade de cor, direção de movimento e direção da iluminação.

Experimentos em Psicologia têm mostrado como explorar as características para realizar tarefas pré-atentivas, tais como:

- **Detecção de alvos:** presença ou ausência de um objeto específico;
- **Verificação de fronteiras:** detectar fronteiras entre dois grupos de elementos quando todos os elementos de cada grupo compartilham uma propriedade visual em comum (cores ou formas);
- **Rastreio de região:** quando os observadores procuram a região onde um ou mais elementos têm uma característica única conforme eles movem no tempo e espaço; e
- **Contagem e estimativa:** quando os usuários contam ou estimam o número de elementos que compartilham uma mesma característica visual.

Atividade 3

- 3.1. Escolha um conjunto de dados de classificação, que contém pelo menos 3 variáveis, para fazer uma análise completa da visualização destes dados com gráficos 2D.
- 3.2. Utilize o mesmo conjunto de dados do item 3.1 para fazer uma análise completa da visualização destes dados com gráficos 3D.
- 3.3. Escolha um conjunto de dados de campos vetoriais 3D ou linhas de fluxo 3D para fazer a representação destes dados com as bibliotecas apresentadas nesta seção.

4. Taxonomia de dados

Na Biologia, taxonomia significa nomear, definir e classificar organismos. Com os conjuntos de dados, podemos utilizar a mesma ideia. A taxonomia representa uma estrutura formal de classes ou tipos de objetos dentro de um domínio de conhecimento, usando atributos e rótulos para facilitar a localização de informações relacionadas.

Definir e usar uma taxonomia pode oferecer benefícios adicionais, pois os usuários do sistema estarão categorizando conteúdos usando um conjunto de rótulos que pode ser considerado como ponto de referência de integração entre diferentes sistemas. Quando olhamos para uma tabela de dados, a última coisa que queremos fazer é ler linha por linha para entendê-la. A taxonomia de dados permite que essa leitura detalhada não precise ser feita.

A taxonomia de dados é a classificação de dados em grupos hierárquicos para criar estrutura, padronizar a terminologia e popularizar um conjunto de dados dentro de uma organização. Os gráficos de taxonomia de dados mostram essa hierarquia usando caixas e linhas e limitam os dados mostrados aos nomes das observações e aos atributos disponíveis.

Os gráficos de taxonomia de dados têm regras flexíveis de design e granularidade, o que significa que podem ser personalizados de acordo com as necessidades de uma empresa. Estes gráficos são muito semelhantes a várias representações de estrutura de dados, incluindo ontologias de dados, hierarquias de dados, metadados, classificações de dados e dicionários de dados.

Deveremos criar regras específicas para classificar ou categorizar qualquer objeto em um domínio. Estas regras devem ser completas, consistentes e inequívocas. Além disso, as regras devem ser bem construídas para garantir que qualquer objeto recém-descoberto se encaixe em uma e apenas uma categoria. Todas as propriedades da classe acima de uma determinada classe são herdadas (conceito de hierarquia). Pode acontecer também que uma classe tenha novas propriedades resultantes da associação de cada objeto em relação aos demais objetos.

Uma taxonomia de dados é uma estrutura hierárquica que separa os dados em classes específicas de dados com base em características comuns. A taxonomia representa uma forma conveniente de classificar os dados para provar que são únicos e sem redundância. Isso inclui elementos de dados primários e gerados a partir dos primários.

As taxonomias são diferentes dos metadados, pois uma taxonomia ajuda você a organizar seu conteúdo em relacionamentos hierárquicos. A classificação de conteúdo e objetos em uma taxonomia pode tornar muito mais fácil pesquisar ou navegar por conteúdos ou sistemas de gerenciamento de conteúdos quando você não tiver certeza do que está procurando.

A utilização da taxonomia pode oferecer muitos benefícios, pois os usuários do sistema estarão categorizando conteúdos usando os rótulos criados. As empresas aplicam taxonomias para:

- melhoria da qualidade de seus dados;
- organização de metadados em um formato de fácil compreensão;
- gerenciamento de conteúdos por meio de sistemas de controle de dados;
- facilitar conferências de informações para um administrador de dados; e
- orientação do aprendizado de máquina para identificar tendências e padrões.

A taxonomia de dados não é o mesmo que um gráfico de taxonomia de dados. A taxonomia em si é o processo de classificação, que não requer registros. No momento em que dois analistas concordam com as categorias, eles constróem uma taxonomia de dados. No entanto, quando eles desenham esta situação em um diagrama de árvore, ele se torna um gráfico de taxonomia de dados.

Na visualização científica, podemos focar na taxonomia para dados, técnicas de visualização, tarefas e métodos de interação. Existem muitas taxonomias propostas para a visualização científica, mas vamos focar nas propostas por Keller e Keller (1994), que classificaram as técnicas de visualização com base nos tipos de dados que são analisados e nas tarefas do usuário.

Os dados são classificados de acordo com seus tipos: Escalar, Nominal, Direcional (ou campo direcional), Forma, Posição e Objeto ou região espacialmente estendida (Keller e Keller, 1994). Algumas tarefas que o usuário pode estar interessado em realizar são as seguintes:

- **Identificação:** estabelece as características para reconhecer um objeto;
- **Localização:** verificação da posição de um objeto;
- **Distinção:** reconhecimento dos objetos;
- **Domínio:** encontrar os possíveis valores que um dado pode ter em um de seus atributos;
- **Categorização:** divisão dos objetos em classes;
- **Agrupamento:** criar os grupos que contém objetos com valores de atributos semelhantes;
- **Rankeamento:** associação dos objetos por uma ordem ou posição relativa;
- **Comparação:** percepção de similaridades e diferenças;
- **Detecção de Anomalias:** encontrar dados que possuem valores muito diferentes aos demais dados;
- **Valor derivado:** aplicar uma operação matemática para obter um valor derivado do conjunto de dados;
- **Associação:** união do que pode ser de um mesmo tipo usando um critério ou relacionamento; e

- **Correlação:** criação de uma conexão direta entre os dados.

Com base nesta lista de tarefas relacionadas à taxonomia, os autores categorizaram mais de 100 técnicas possíveis. Em relação aos tipos de dados, temos a seguinte lista proposta por Shneiderman (1996):

- **Unidimensionais:** são os itens que estão organizados de maneira sequencial (textos, números, códigos de programas de computador). As tarefas de consulta destes dados são feitas com base em algum atributo escolhido.

- **Bidimensionais:** são itens do conjunto de dados que cobrem uma área (mapas geográficos, malhas geométricas, poligonais). As tarefas de consulta destes dados envolvem a determinação de itens adjacentes, caminhos, filtros e detalhes sobre os dados.

- **Tridimensionais:** são itens que possuem um volume ou um relacionamento mais complexo com outros dados (objetos, partes do corpo humano, construções, modelos 3D, poliedros). As tarefas de consulta destes dados envolvem a compreensão de posição, orientação e até mesmo a resolução de problemas de oclusão.

- **Temporal:** são os itens que possuem um tempo de início e término (séries temporais, eletrocardiogramas, gerenciamento de projetos). As tarefas de consulta destes dados envolvem eventos anteriores e posteriores de um momento específico. Além disso, podem ser feitas consultas de períodos específicos pertencentes ao conjunto de dados.

- **Multidimensionais:** são os itens com n atributos, ou seja, pertencem ao espaço n -dimensional, considerando $n > 3$. A maioria dos conjuntos de dados relacionais é deste tipo. As tarefas de busca incluem a determinação de grupos, padrões, correlações entre variáveis, outliers e limpeza de dados.

- **Árvores:** são os dados que possuem alguma ligação com um item ‘pai’, exceto dados raiz (dados hierárquicos, estruturas em árvore, ConeTree, TreeMap). As tarefas de buscas destes dados incluem as reconstruções das ramificações que geraram os itens por meio dos seguintes questionamentos:

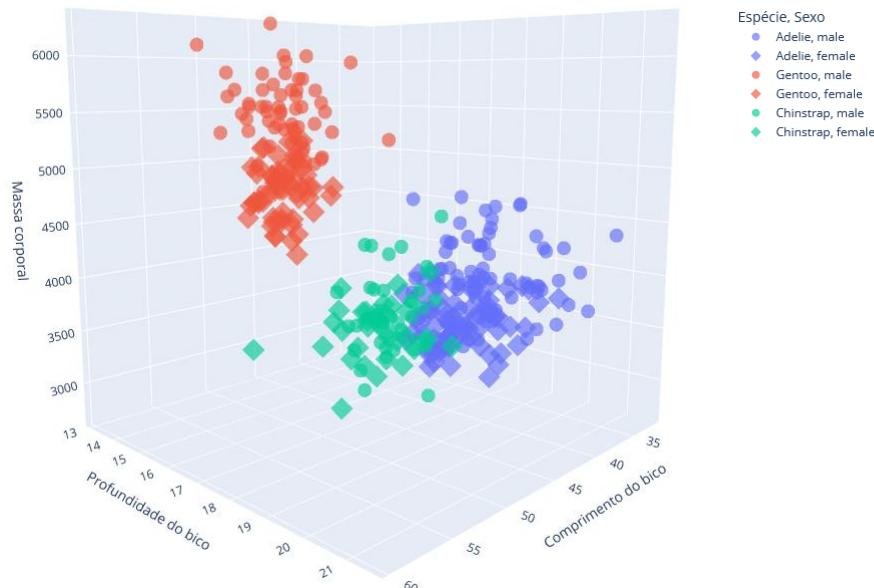
- quantos níveis a árvore possui?
- quantos ‘filhos’ um item possui?
- quantos itens estão em um mesmo nível?

- **Redes:** são dados que não têm relações que podem ser representadas como árvores (redes neurais artificiais, redes sociais, redes de computadores). As tarefas de busca nestes dados incluem as descobertas de ligações aos itens, medidas sobre os nós da rede e caminhos de menor ou de maior custo.

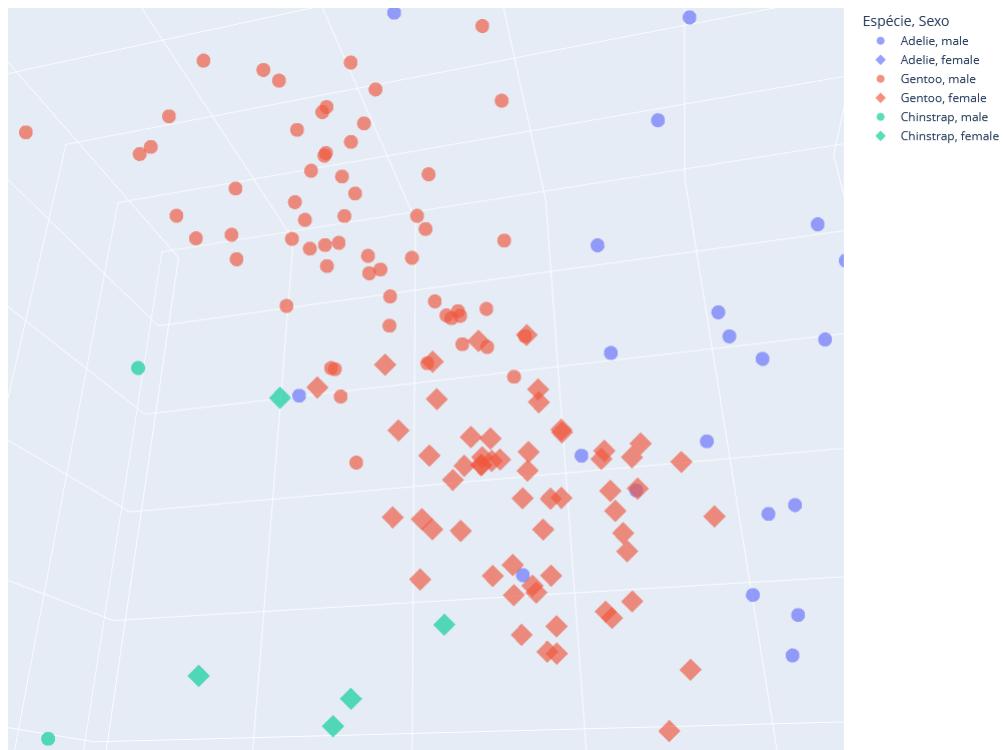
A área de visualização de dados e informações é relativamente nova e vem ganhando destaque, pois com os avanços tecnológicos e científicos, a quantidade de dados e informações que os usuários podem coletar se torna, muitas vezes, enorme, podendo sobrecarregá-los. Logo, o usuário deve saber como organizar seus dados para que eles possam ser corretamente apresentados e interpretados.

As tarefas dos usuários para a visualização de dados são responsáveis na determinação de novos mecanismos de interação das ferramentas de visualização em um projeto. De acordo com Shneiderman (1996), os usuários podem realizar as seguintes tarefas para tentar extrair conhecimento dos dados:

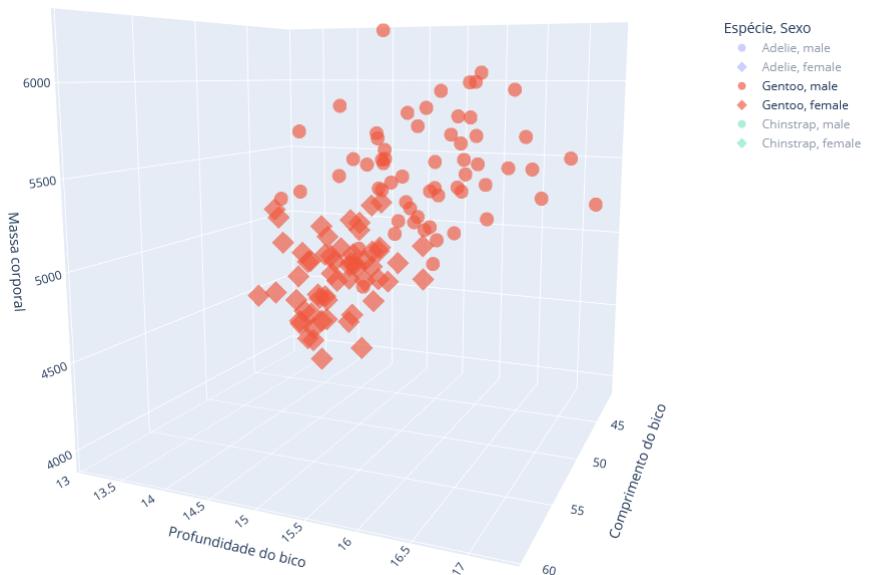
- **Visão Geral:** obter uma visão geral de todo o conjunto de dados.



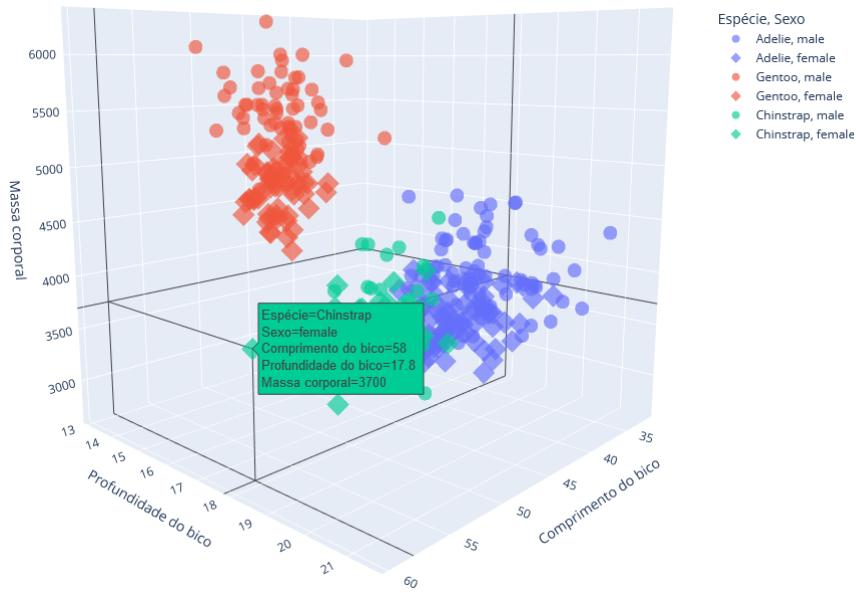
- **Zoom:** aproximar a visualização para os itens de interesse para obter mais detalhes. Em algumas situações, podemos diminuir a aproximação para visualizar subconjuntos de dados.



- **Filtro:** esconder alguns itens para reduzir o tamanho do espaço de busca do usuário e selecionar dados de interesse para satisfazer determinadas condições. Os itens que não são selecionados pelo filtro podem ficar escondidos ou ficam com opacidade reduzida, ou seja, perdem o destaque na visualização dos dados.



- **Detalhes sob demanda:** selecionar um item ou grupos e obter detalhes quando for necessário.



Open Data for the Northern Tornadoes Project

Tornado Worst Point

Membro Privado

Western University

Resumo

This feature layer contains centrelines, including start and end points, and worst points of damage for tornado events, and extents of damage and worst points of damage for downburst events.

[Visualizar Detalhes Completos](#)

Detalhes

Conjunto de Dados
Feature Layer

11 de maio de 2021
Informação Atualizada

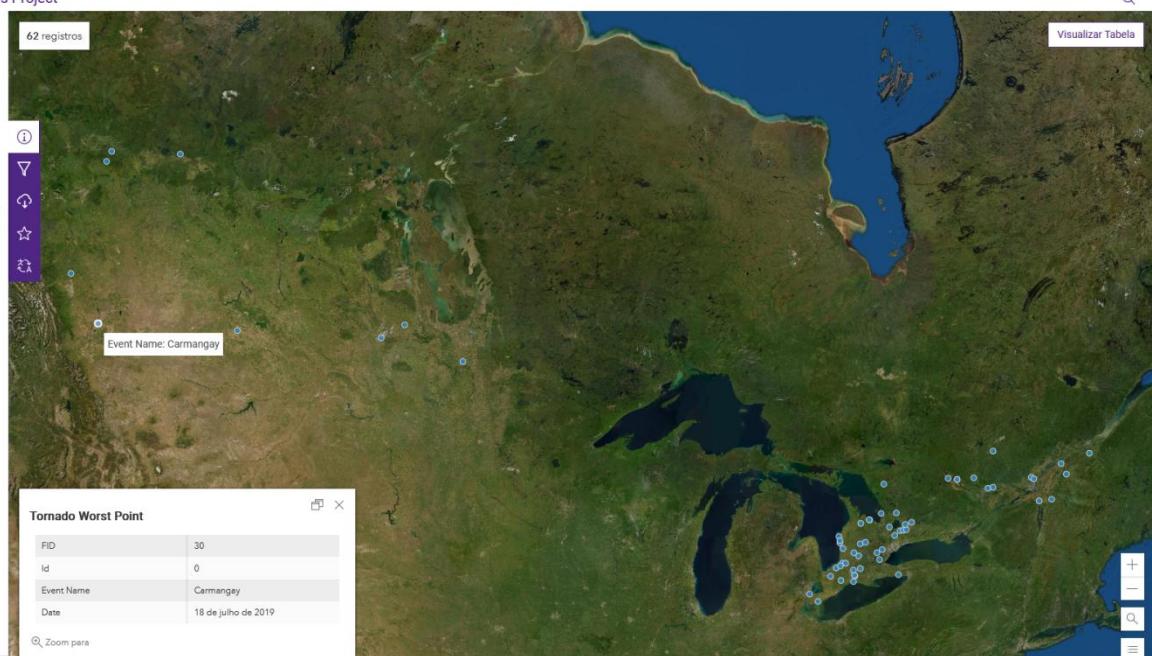
26 de janeiro de 2022
Dados Atualizados

29 de abril de 2021
Data de Publicação

62 Registros
[Visualizar tabela de dados](#)

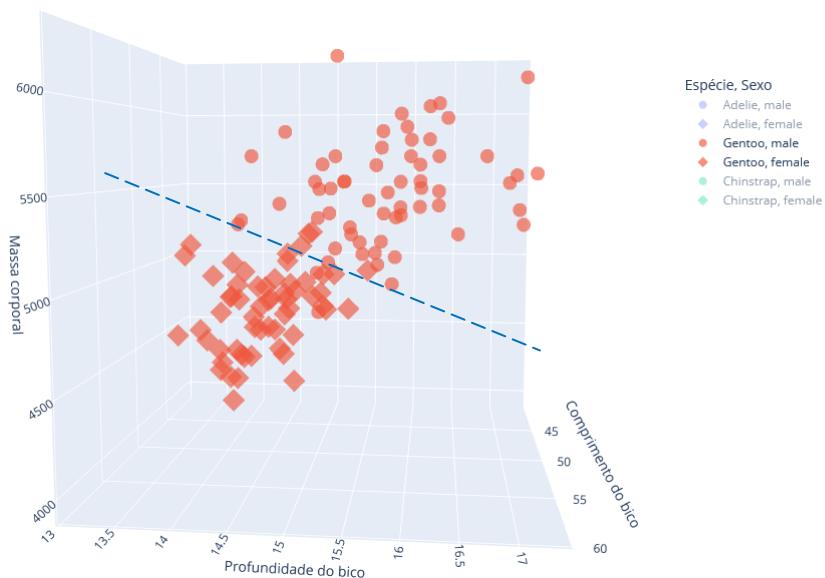
Público
Qualquer pessoa pode ver este conteúdo

Licença CC BY-NC
[Visualizar detalhes da licença](#)



Disponível em: <https://ntpopendata-westernu.opendata.arcgis.com/datasets/westernu::tornado-worst-point>

- **Relacionar:** encontrar alguma espécie de relação entre dados de um conjunto, após análise de dois ou mais atributos ou agrupamentos.



- **Histórico:** manter um histórico e permitir retroceder, refazer e refinar um processo de exploração.

[Home](#) [Browser](#) [Citing SO](#) [GFF3](#) [GVF](#) [Software](#) [Request A Term](#) [Issues](#)

Search for Terms: Select Release:

[Term Only](#) as [OBO Format](#) [Export](#)

double_stranded_cDNA (CURRENT_RELEASE)

SO Accession:	SO:0000758 (SOWIK)
Definition:	DNA synthesized from RNA by reverse transcriptase that has been copied by PCR to make it double stranded.
Synonyms:	double strand cDNA, double-strand cDNA, double stranded cDNA
Parent:	cDNA (SO:0000756)

In the image below graph nodes link to the appropriate terms. Clicking the image background will toggle the image between large and small formats.

```

graph TD
    DS[double stranded cDNA] --> cDNA[cDNA]
    cDNA --> DNA[DNA]
    DNA --> NA[nucleic acid]
    NA --> PA[polymer_attribute]
    PA --> SA[sequence_attribute]
  
```

sequence_attribute

- feature_attribute
- nucleic_attribute
- nucleic_acid
 - DNA
 - cDNA
 - double_stranded_cDNA
 - single_stranded_cDNA
 - genomic_DNA
- GNA
- LNA
- PNA
- RNA
- TNA
- morpholino_backbone
- peptidyl
- synthetic_sequence
- topology_attribute
- sequence_location
- sequence_source
- variant_quality
- sequence_collection
- sequence_feature
- sequence_variant

Obsolete Terms

Relationship

Disponível em: http://www.sequenceontology.org/browser/current_release/term/SO:0000758

- **Extrair:** encontrar e acessar atributos dos dados de um determinado conjunto, além de retirar itens ou dados em um formato que possa facilitar outras tarefas de análise ou visualização por meio de outras ferramentas.

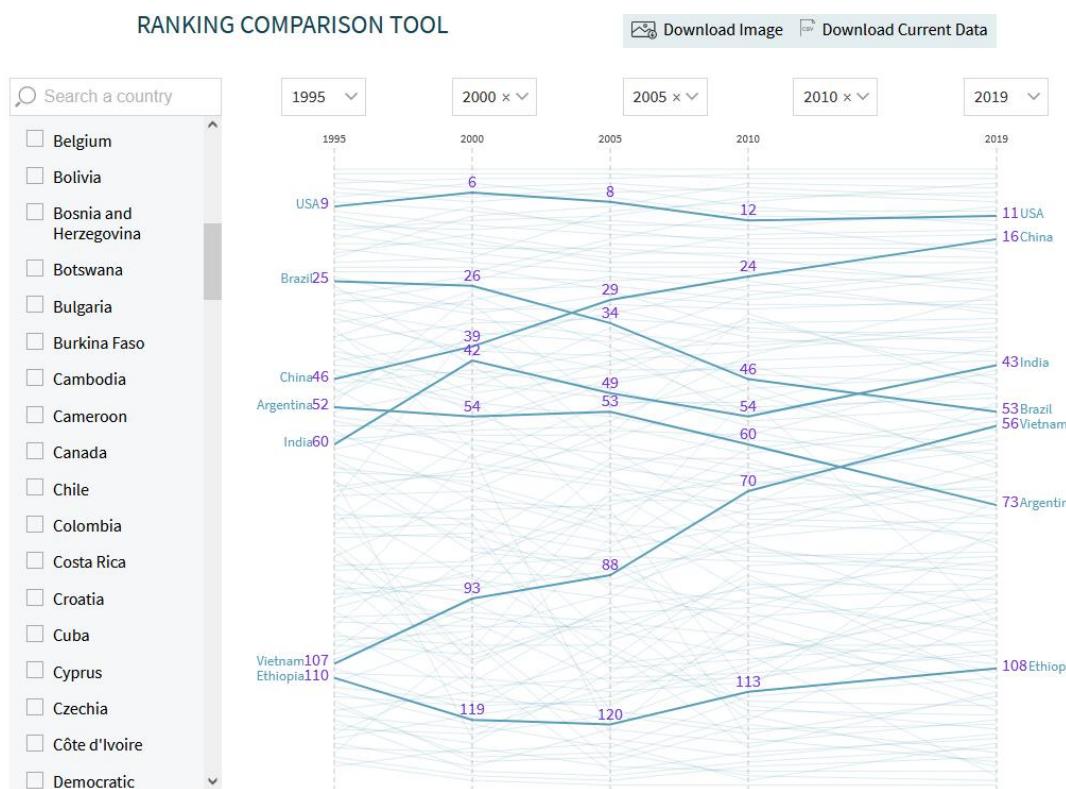
Um sistema eficiente de visualização permite que todas ou a maioria destas tarefas seja feita de maneira simples. Os sistemas de visualização podem ser classificados de acordo com os tipos de dados, métodos de interação e técnicas de visualização (Keim, 2002).

Em relação aos tipos de dados, temos os seguintes tipos de informações:

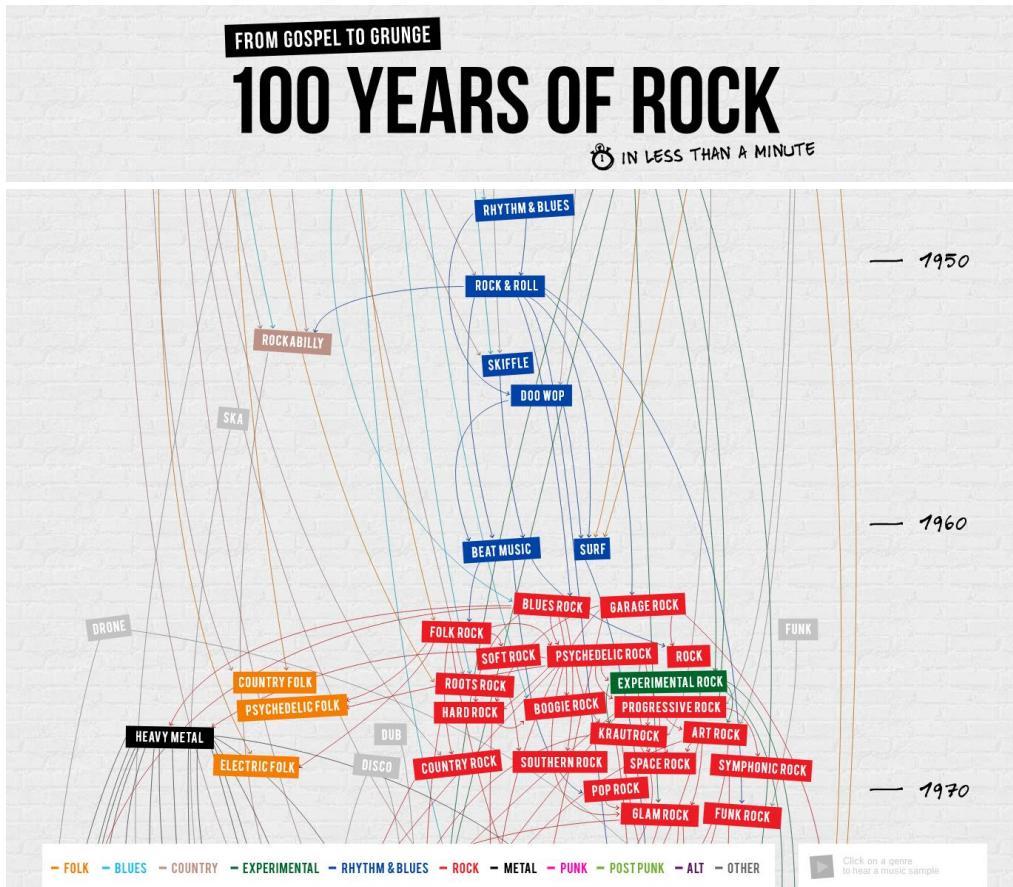
- **Unidimensional:** dados temporais, dados financeiros de preços (ações, dólar, inflação);
- **Bidimensional:** mapas, gráficos;
- **Multidimensional:** tabelas de bancos de dados, planilhas;
- **Texto e hipertexto:** análise de documentos, dados da internet;
- **Hierarquias e grafos:** redes de informações, hierarquias, sistemas dinâmicos; e
- **Algoritmos e Softwares:** softwares, dados sobre memória, trechos de execução.

Em relação aos métodos de interação, temos os seguintes exemplos:

- **Projeção dinâmica:** seleção de variáveis para comparação com outras já mostradas em um gráfico.

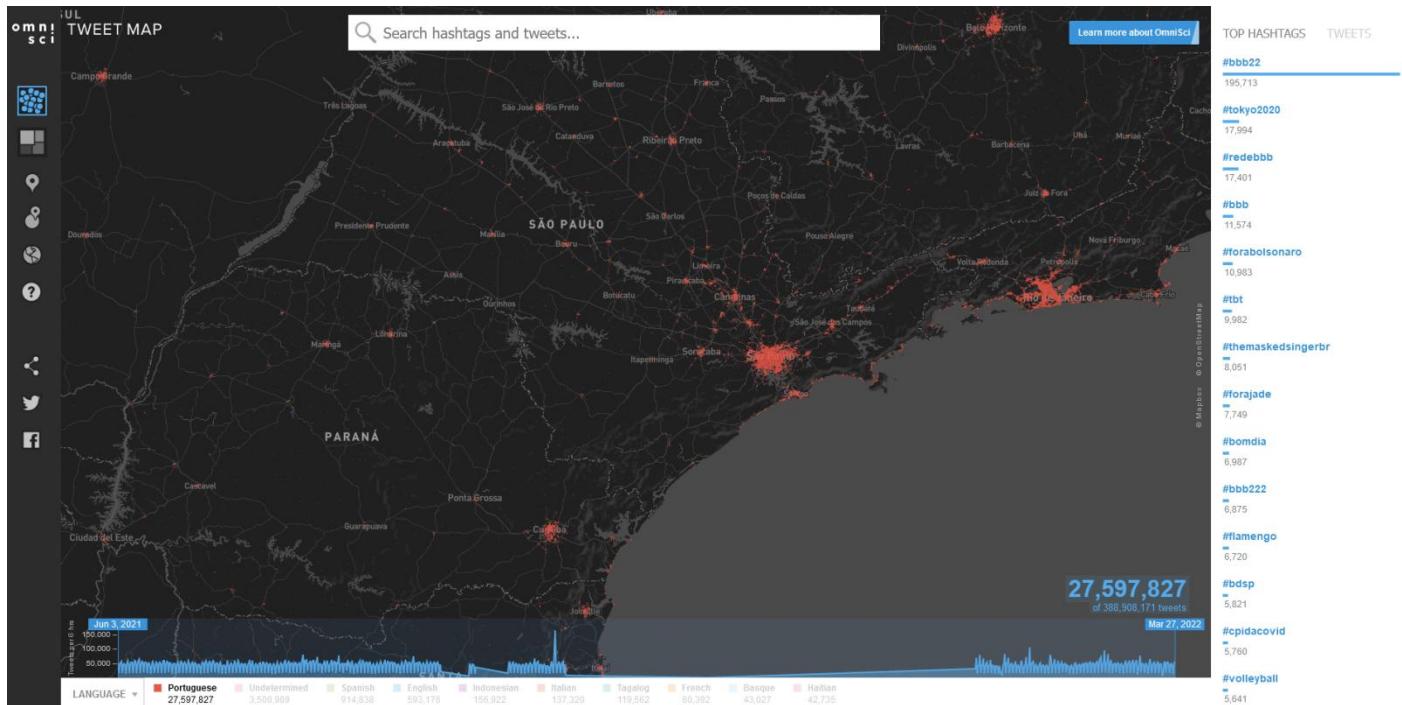


Disponível em: <https://atlas.cid.harvard.edu/rankings>



Disponível em: <https://www.concerthotels.com/100-years-of-rock/>

- **Filtro interativo:** trata-se da seleção direta ou por consulta para filtrar elementos e grupos sem afetar o restante da visualização dos dados. Na área de visualização científica, podemos definir a interação como um mecanismo para modificar o que e como os usuários veem a representação visual e os dados.



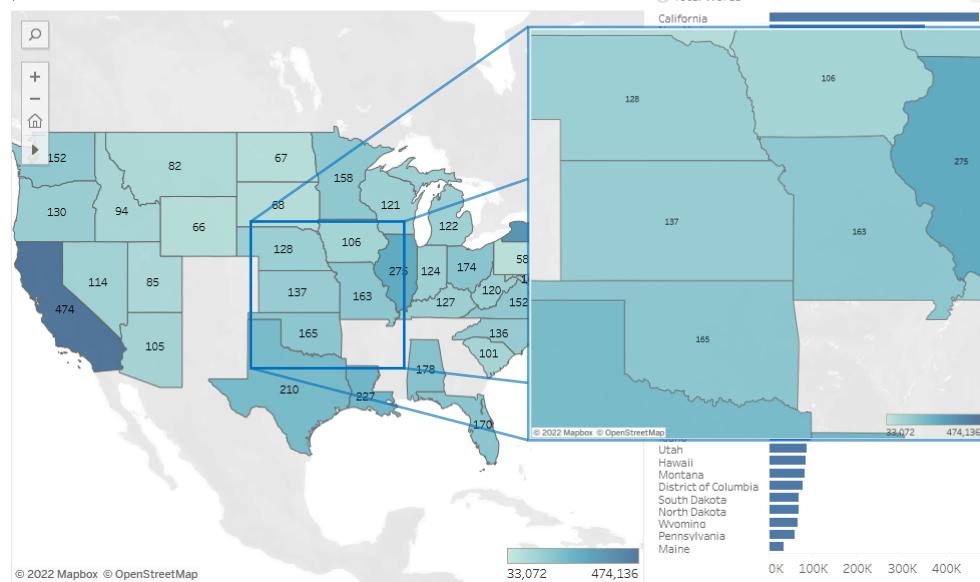
Disponível em: <https://www.heavy.ai/demos/tweetmap>

- **Zoom interativo:** a aproximação (zoom in) ou o afastamento (zoom out) que auxiliam na compreensão dos conjuntos de dados. Além desta aproximação, em alguns casos o usuário pode interagir com a visualização fazendo

rotações das áreas representadas (por exemplo, com mapas interativos podemos mudar a orientação para visualizar as regiões selecionadas).

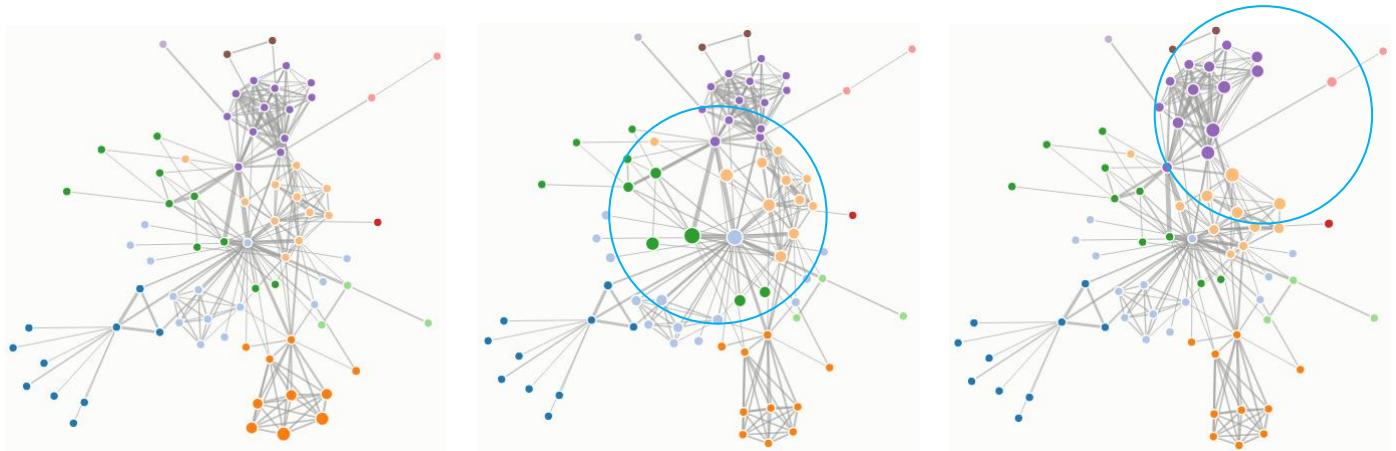
State Statutes, 2021

2021 marks the first year of US state statute data for the State Regdata Project. Data series for this data set include; restriction counts, word counts, conditional count, Flesch reading ease score, sentence length, shannon entropy score, and industry relevance probabilties.



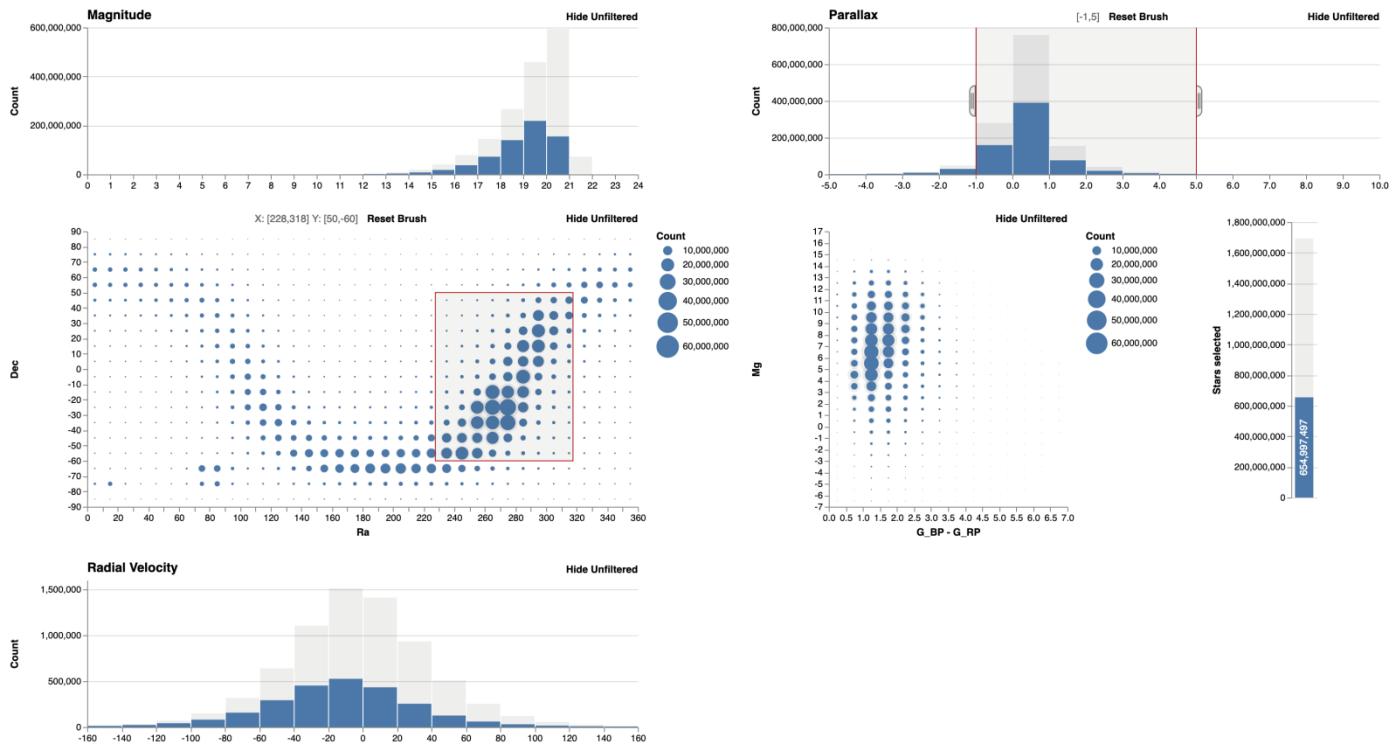
Disponível em: <https://www.quantgov.org/state-statutes>

- **Distorção interativa:** exibe detalhes de partes específicas da visualização, enquanto o usuário não perde a visão geral. Layouts com a técnica fisheye, radial ou hiperbólico envolvem a transformação de pixels em áreas selecionadas pelo usuário. As regiões de pixels são aumentadas ou reduzidas para exibir um detalhamento seletivo dos dados representados.



Disponível em: <https://bostocks.org/mike/fisheye/>

- **Linking and brushing interativo:** a ideia deste tipo de visualização é de combinar diferentes métodos de visualização para superar as deficiências de técnicas únicas. As alterações interativas feitas em uma visualização são refletidas automaticamente nas outras visualizações. Conectando várias visualizações por meio desta técnica, fornecemos mais informações para o usuário do que considerar as visualizações de componentes de forma independente. O conjunto de visualizações que permite esse tipo de interação forma um sistema baseado em Múltiplas Visões Coordenadas.



Disponível em: <https://github.com/vega/falcon>

4.1. Outras representações gráficas

Agora vamos aprender alguns tipos de gráficos e técnicas mostrados nesta seção. Começando pelo gráfico de radar, estrela ou “teia de aranha”, temos um estilo de visualização amplamente usado para comparar pessoas, lugares ou outras entidades utilizando várias métricas. As métricas nos dados precisam compartilhar a mesma escala, então você verá mais comumente radares usados para pontos percentuais ou pontuações.

Para cada métrica temos um ponto marcado no eixo do radar. As arestas são construídas quando unimos as métricas ‘vizinhas’ de um dado, formando um polígono que representa as informações deste dado. As comparações entre os dados podem ser feitas por sobreposição dos polígonos ou plotando cada dado com seu respectivo polígono separadamente. Podemos usar os dados de uma planilha, com as informações de uma coluna para cada métrica. Veja o exemplo mostrado a seguir com os dados do conjunto The Beatles, com 5 métricas (rótulos).

Beatle	Composition	Voice	Rhythm	Solos	Humour
John	10	8	6	5	9
Paul	10	8	6	7	6
George	8	7	6	10	5
Ringo	2	2	10	5	5

```

import plotly.io as pio
pio.renderers
pio.renderers.default = 'browser'
import plotly.graph_objects as go

rotulos = ['Composition', 'Vocal', 'Rhythm', 'Solos', 'Humour']

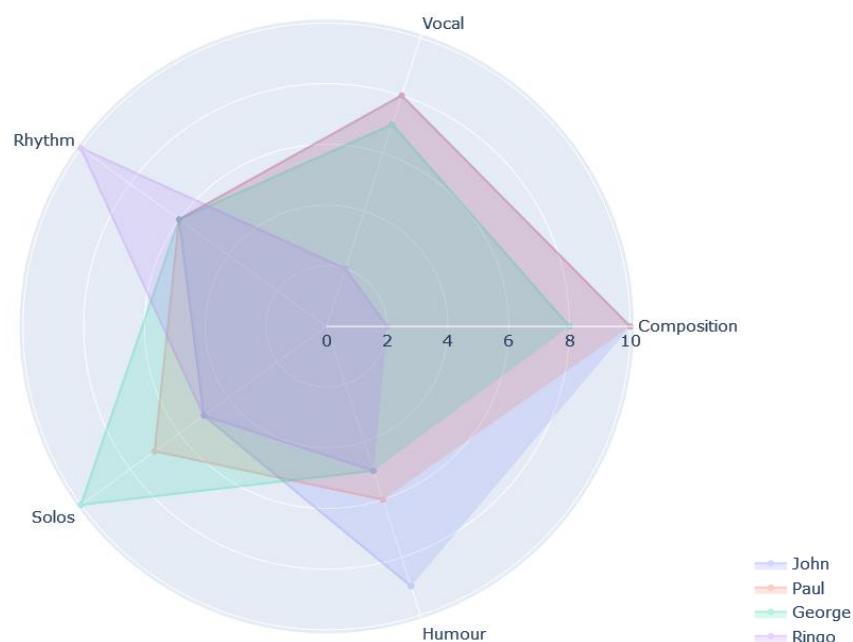
fig = go.Figure()

fig.add_trace(go.Scatterpolar(r = [10, 8, 6, 5, 9], theta = rotulos, fill = 'toself',
                             name = 'John', opacity = 0.3))
fig.add_trace(go.Scatterpolar(r = [10, 8, 6, 7, 6], theta = rotulos, fill = 'toself',
                             name = 'Paul', opacity = 0.3))
fig.add_trace(go.Scatterpolar(r = [8, 7, 6, 10, 5], theta = rotulos, fill = 'toself',
                             name = 'George', opacity = 0.3))
fig.add_trace(go.Scatterpolar(r = [2, 2, 10, 5, 5], theta = rotulos, fill = 'toself',

```

```
name = 'Ringo', opacity = 0.3))

fig.show()
```



Dados disponíveis em: <https://app.flourish.studio/@flourish/radar>

A visualização feita com gráficos em coordenadas paralelas é usada para plotar dados numéricos multivariados. Os gráficos de coordenadas paralelas são ideais para comparar muitas variáveis e ver as relações entre elas. Por exemplo, se você tivesse que comparar uma série de produtos com os mesmos atributos.

Em um gráfico de coordenadas paralelas, cada variável recebe seu próprio eixo e todos os eixos são colocados em paralelo entre si. Cada eixo pode ter uma escala diferente, pois cada variável trabalha com uma unidade de medida diferente, ou todos os eixos podem ser normalizados para manter todas as escalas uniformes. Os valores são plotados como uma série de linhas conectadas em todos os eixos. Isso significa que cada linha é uma coleção de pontos colocados em cada eixo, que foram todos conectados.

A ordem em que os eixos são organizados pode afetar a maneira como o leitor entende os dados. Uma razão para isso é que as relações entre variáveis adjacentes são mais fáceis de perceber do que para variáveis não adjacentes. Portanto, reordenar os eixos pode ajudar a descobrir padrões ou correlações entre variáveis. Quando reorganizamos os dados, podemos obter diferentes visões ou perspectivas dos dados representados. Ao reordenar eixos ou linhas em visualizações baseadas em tabelas, podemos encontrar correlações entre os atributos.

A desvantagem dos gráficos de coordenadas paralelas é que eles podem ficar sobrecarregados e, portanto, ilegíveis quando são muito densos em dados. No exemplo mostrado a seguir, temos os dados do conjunto de pinguins em coordenadas paralelas da biblioteca **plotly**. A função **Parcoords** permite que o usuário selecione intervalos de valores nos eixos, criando as ligações com as variáveis de outros eixos. Além disso, o usuário pode alterar as ordens dos eixos plotados.

```
import pandas as pd
import plotly.io as pio
import plotly.graph_objects as go

pio.renderers
pio.renderers.default = 'browser'

df = pd.read_csv('C:/dados/penguin2.csv')

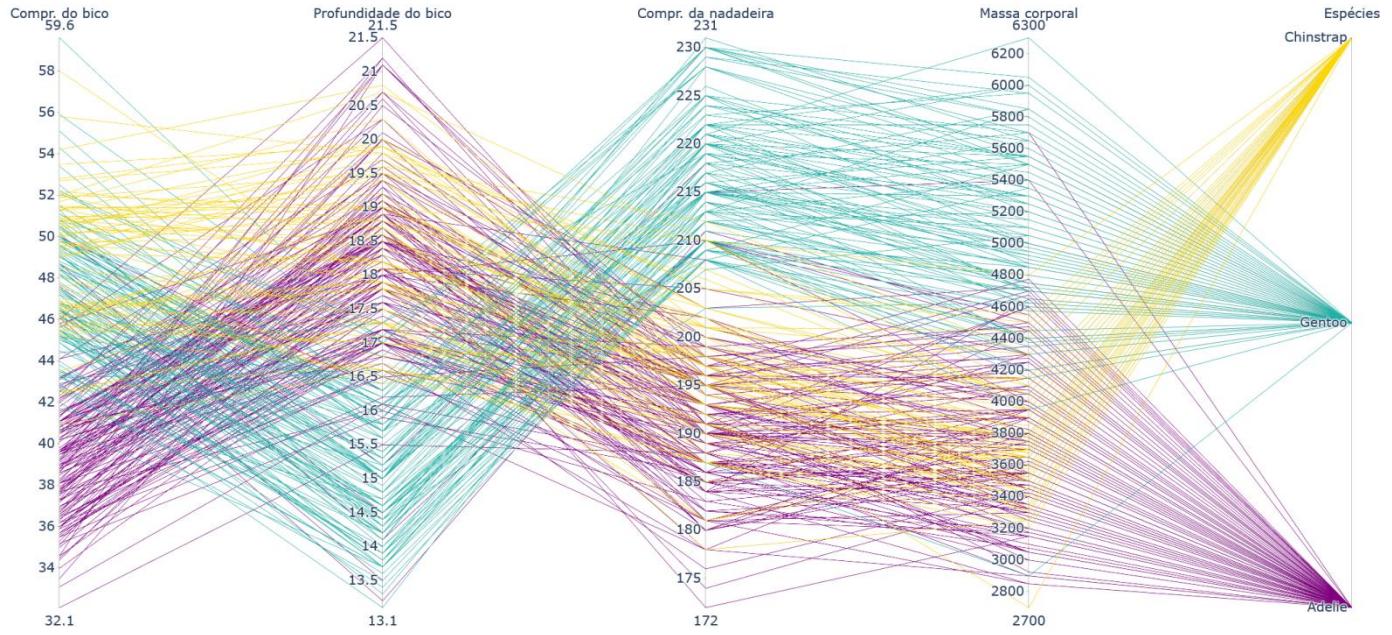
fig = go.Figure(data = go.Parcoords(line = dict(color = df['Cor_id']),
                                      colorscale = [[0,'purple'],[0.5,'lightseagreen'],[1,'gold']]),
                  dimensions = list([dict(label = 'Compr. do bico', values = df['Comprimento do bico']),
                                     dict(label = 'Profundidade do bico', values = df['Profundidade do bico']),
```

```

dict(label = 'Compr. da nadadeira', values = df['Comprimento da nadadeira']),
dict(label = 'Massa corporal', values = df['Massa corporal']),
dict(label = 'Espécies', values = df['Cor_id'], tickvals = [1,2,3],
     ticktext = ['Adelie', 'Gentoo', 'Chinstrap']))))
fig.update_layout(font = dict(size = 24))

fig.show()

```



Utilizando a biblioteca **bokeh**, podemos construir gráficos que permitem a interação com o usuário para selecionar dados de um gráfico para mostrá-los em outro gráfico. A função `ColumnDataSource` serve de ligação dos dados do gráfico `p1`, que contém os dados dos pinguins com 2 atributos, com os dados do gráfico `p2`, que contém os dados de outros dois atributos dos pinguins.

```

from bokeh.layouts import gridplot
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.transform import factor_cmap, factor_mark
import pandas as pd

df = pd.read_csv('C:/dados/penguin2.csv')
especies = ['Adelie', 'Gentoo', 'Chinstrap']
markers = ['hex', 'circle_x', 'triangle']

tools = 'box_select,lasso_select,box_zoom,pan'

source = ColumnDataSource(data = dict(x = df.loc[:, 'Comprimento do bico'],
                                       y = df.loc[:, 'Profundidade do bico'],
                                       z = df.loc[:, 'Comprimento da nadadeira'],
                                       w = df.loc[:, 'Massa corporal'],
                                       esp = df.loc[:, 'Espécie']))

p1 = figure(tools = tools, title = None)
p1.xaxis.axis_label = 'Comprimento do bico'
p1.yaxis.axis_label = 'Profundidade do bico'
p1.scatter(x = 'x', y = 'y', source = source, legend_field = 'esp', fill_alpha = 0.4, size = 12,
           marker = factor_mark('esp', markers, especies),
           color = factor_cmap('esp', 'Category10_3', especies))
p1.legend.location = 'bottom_right'

p2 = figure(tools = tools, title = None)
p2.xaxis.axis_label = 'Comprimento da nadadeira'
p2.yaxis.axis_label = 'Massa corporal'
p2.scatter(x = 'z', y = 'w', source = source, legend_field = 'esp', fill_alpha = 0.4, size = 12,
           marker = factor_mark('esp', markers, especies),

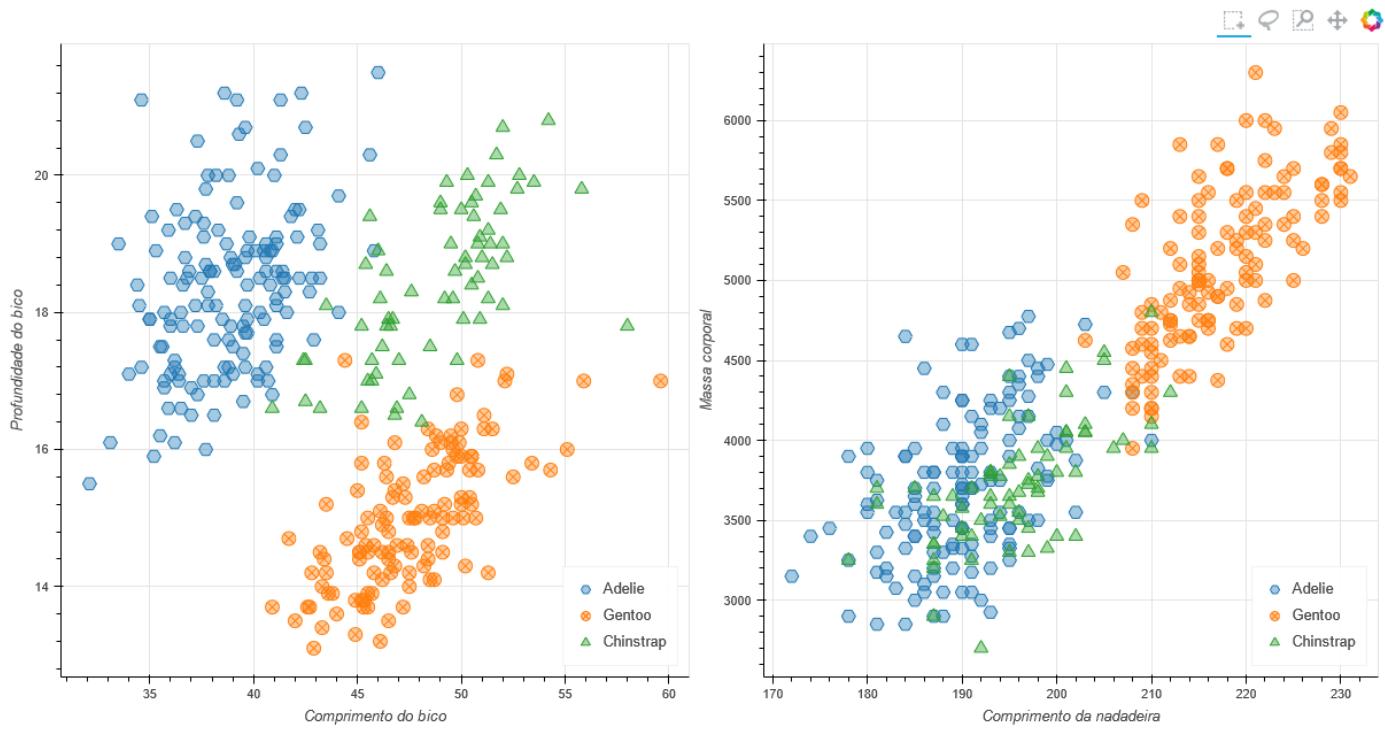
```

```

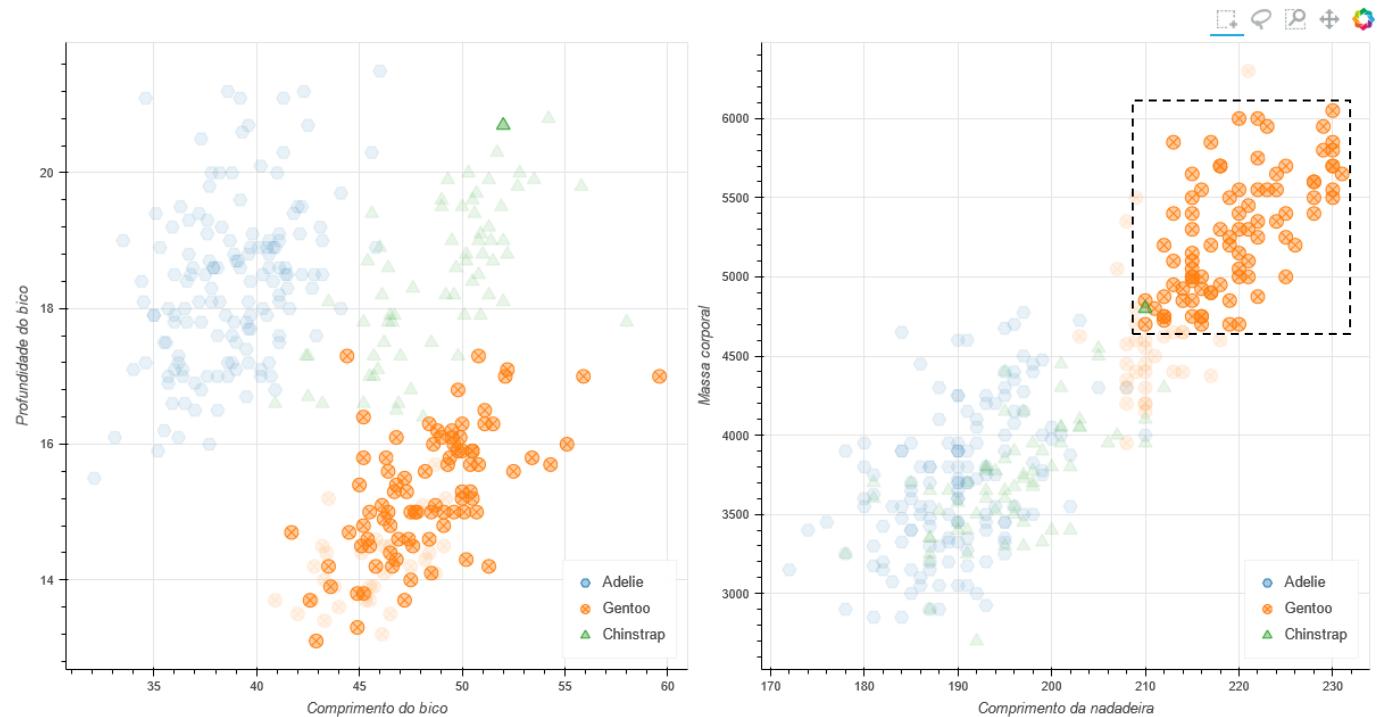
color = factor_cmap('esp', 'Category10_3', species))
p2.legend.location = 'bottom_right'

p = gridplot([[p1, p2]])
show(p)

```

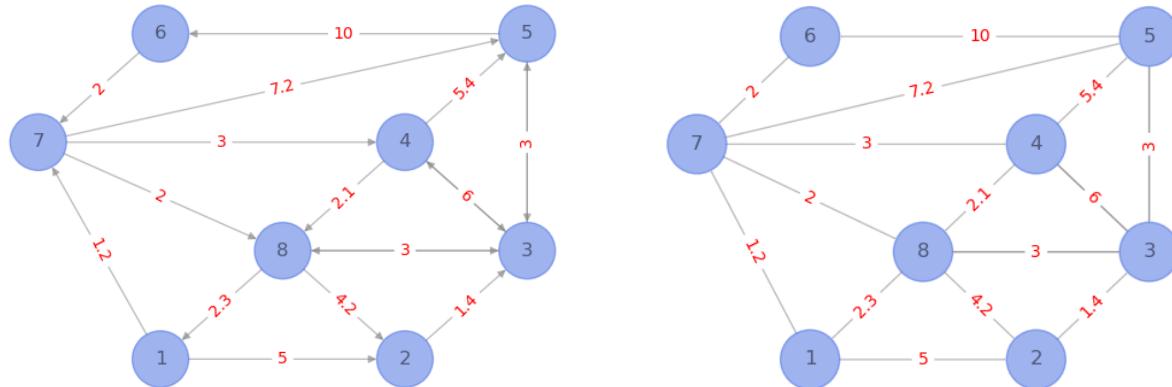


Quando selecionamos uma parte do conjunto de dados de um gráfico, os mesmos dados são mostrados em destaque no outro gráfico.

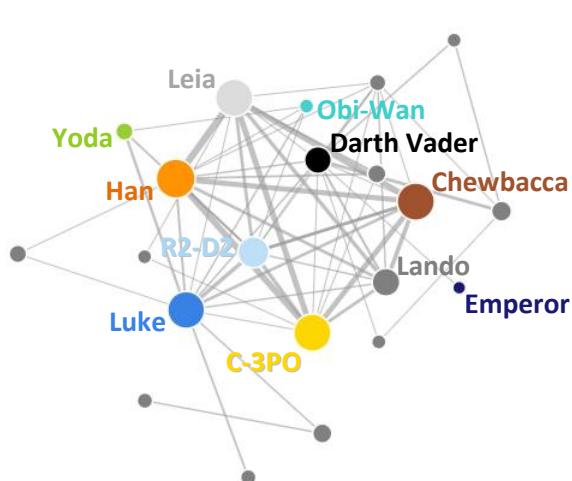


Os conjuntos de dados podem conter relações entre os atributos. As informações sobre os relacionamentos entre os atributos podem ser representadas de diferentes maneiras: conjuntos e subconjuntos, hierarquia (pai ou filho), ou conectividades (redes de computadores, cidades ou localizações). Uma forma de representar o relacionamento dos atributos é por meio de Grafos, que são definidos como conjuntos compostos por vértices ou nós e seus relacionamentos são feitos por meio de arestas.

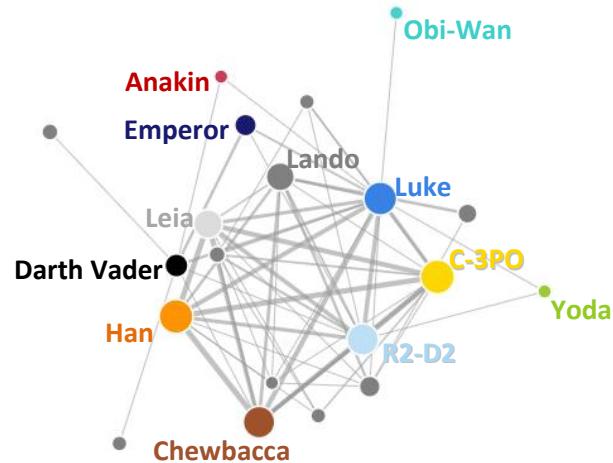
Um grafo pode ser direcionado ou não direcionado, conter pesos entre os nós ou simplesmente mostrar as possíveis ligações entre os nós sem pesos. Quando os nós representam locais, suas representações devem ser feitas usando suas respectivas coordenadas x e y. Em outros tipos de grafos, os nós podem ocupar qualquer posição no plano de coordenadas x e y.



No exemplo mostrado a seguir, temos a representação da rede social dos personagens do Star Wars em dois episódios. As conexões foram feitas com os personagens que conversaram entre si por meio de um grafo com apenas nós e arestas, sem a necessidade de pesos para as ligações.



Episode V: The Empire Strikes Back



Episode VI: Return of the Jedi

Disponível em: <http://evelinag.com/blog/2015/12-15-star-wars-social-network/index.html#.VnAhsTZZG6A>

Usando a biblioteca **networkx**, podemos representar as visualizações de dados de grafos e redes. No primeiro exemplo, vamos mostrar as funções para um exemplo de grafo com as representações dos tempos (em horas) das viagens entre algumas cidades europeias.

Começamos determinando o vetor de arcos, e vamos considerar como hipótese que se trata de um grafo direcionado (DiGraph). Desta forma, o vetor de arcos tem algumas representações de arcos de ida e volta: `['Madrid', 'Paris']`, `['Madrid', 'Bern']`, `['Bern', 'Madrid']`, `['Bern', 'Amsterdam']`, `['Bern', 'Berlin']`, `['Bern', 'Rome']`, `['Amsterdam', 'Berlin']`, `['Amsterdam', 'Copenhagen']`, `['Berlin', 'Copenhagen']`, `['Berlin', 'Budapest']`, `['Berlin', 'Warsaw']`, `['Berlin', 'Rome']`, `['Budapest', 'Warsaw']`, `['Budapest', 'Rome']`, `['Budapest', 'Athens']`, `['Budapest', 'Bucharest']`, `['Bucharest', 'Athens']`, `['Bucharest', 'Ankara']`,

```
import networkx as nx
import matplotlib.pyplot as plt

arcos = [['Madrid', 'Paris'], ['Madrid', 'Bern'], ['Bern', 'Madrid'], ['Bern', 'Amsterdam'],
          ['Bern', 'Berlin'], ['Bern', 'Rome'], ['Amsterdam', 'Berlin'], ['Amsterdam', 'Copenhagen'],
          ['Berlin', 'Copenhagen'], ['Berlin', 'Budapest'], ['Berlin', 'Warsaw'], ['Berlin', 'Rome'],
          ['Budapest', 'Warsaw'], ['Budapest', 'Rome'], ['Budapest', 'Athens'],
          ['Budapest', 'Bucharest'], ['Bucharest', 'Athens'], ['Bucharest', 'Ankara'],
```

```

['Bucharest', 'Kiev'], ['Ankara', 'Moscow'], ['Kiev', 'Moscow'], ['Warsaw', 'Moscow'],
['Moscow', 'Kiev'], ['Warsaw', 'Kiev'], ['Paris', 'Amsterdam'], ['Paris', 'Bern'])

g = nx.DiGraph()
g.add_edges_from(arcos)
plt.figure()

pos = {'Madrid': [36, 0], 'Paris': [114, 151], 'Bern': [184, 116], 'Berlin': [261, 228],
       'Amsterdam': [151, 222], 'Rome': [244, 21], 'Copenhagen': [247, 294],
       'Budapest': [331, 121], 'Warsaw': [356, 221], 'Athens': [390, -44],
       'Bucharest': [422, 67], 'Ankara': [509, -13], 'Kiev': [480, 177], 'Moscow': [570, 300]}

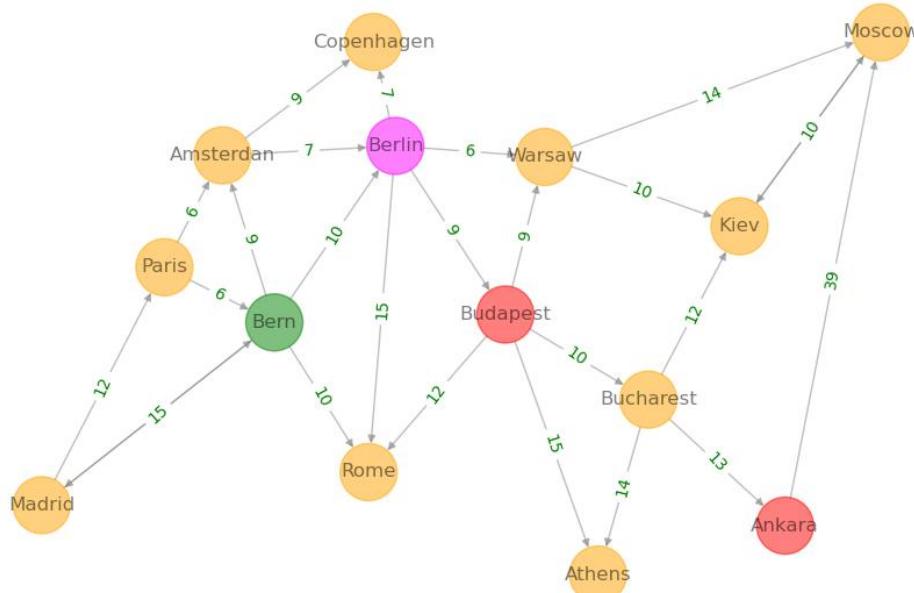
cor = ['orange', 'orange', 'green', 'orange', 'magenta', 'orange', 'orange', 'red', 'orange',
       'orange', 'orange', 'red', 'orange', 'orange']

rotulos = {('Madrid', 'Paris'): '12', ('Madrid', 'Bern'): '15', ('Bern', 'Amsterdam'): '9',
           ('Bern', 'Berlin'): '10', ('Bern', 'Rome'): '10', ('Paris', 'Bern'): '6',
           ('Amsterdam', 'Berlin'): '7', ('Paris', 'Amsterdam'): '6', ('Amsterdam', 'Copenhagen'): '9',
           ('Berlin', 'Copenhagen'): '7', ('Berlin', 'Budapest'): '9', ('Berlin', 'Warsaw'): '6',
           ('Berlin', 'Rome'): '15', ('Budapest', 'Warsaw'): '9', ('Budapest', 'Rome'): '12',
           ('Budapest', 'Bucharest'): '10', ('Budapest', 'Athens'): '15', ('Bucharest', 'Athens'): '14',
           ('Bucharest', 'Ankara'): '13', ('Ankara', 'Moscow'): '39', ('Bucharest', 'Kiev'): '12',
           ('Warsaw', 'Kiev'): '10', ('Warsaw', 'Moscow'): '14', ('Moscow', 'Kiev'): '10'}

nx.draw(g, pos, with_labels = True, node_color = cor, edge_color = 'grey', alpha = 0.5,
        linewidths = 1, node_size = 1250, labels = {node: node for node in g.nodes()})
nx.draw_networkx_edge_labels(g, pos, edge_labels = rotulos, font_color = 'green')

plt.show()

```



Dados disponíveis em: <https://mathspace.co/textbooks/syllabuses/Syllabus-810/topics/Topic-18092/subtopics/Subtopic-246331/>

Para representar os dados do problema de encontrar um circuito Hamiltoniano com 1.000 vértices, podemos utilizar a função `from_pandas_edgelist`, que lê os arcos de um nó (source) para outro nó (target). A representação mostrada a seguir tem o caminho de um circuito encontrado com os arcos indicando as direções dos arcos e os arcos em cinza claro que representam os possíveis arcos desta instância.

```

import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

plt.figure()

df = pd.read_csv('C:/dados/alb1000.csv')

```

```

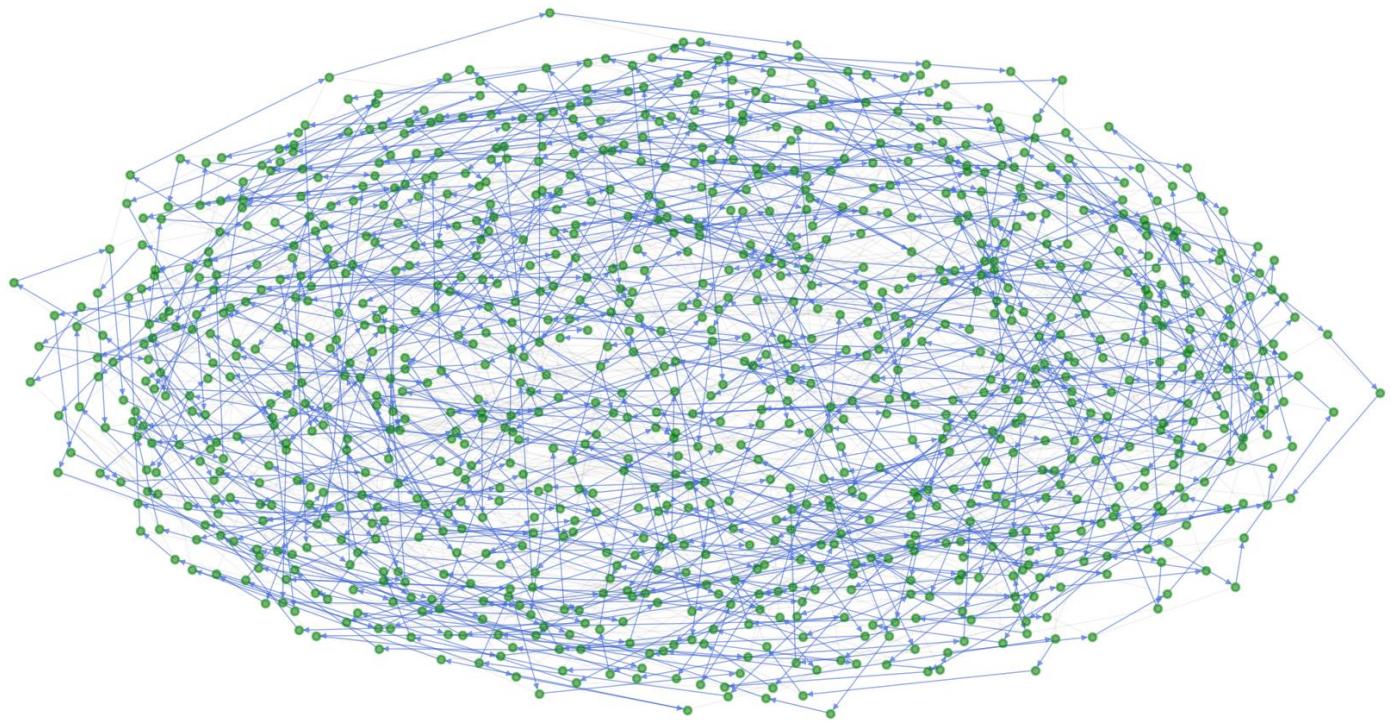
df_s = pd.read_csv('C:/dados/alb1000_opt.csv')

g = nx.from_pandas_edgelist(df, source = 'v1', target = 'v2')
pos = nx.kamada_kawai_layout(g)
nx.draw(g, pos = pos, node_color = 'grey', edge_color = 'grey', alpha = 0.1, linewidths = 0.2,
        node_size = 40)

g1 = nx.from_pandas_edgelist(df_s, source = 'v1', target = 'v2', create_using = nx.DiGraph)
nx.draw(g1, pos = pos, node_color = 'green', edge_color = 'royalblue', alpha = 0.5,
        linewidths = 2, node_size = 40)

plt.show()

```



Dados disponíveis em: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/hcp/index.html>

Nas representações dos dados de problemas de roteamento de veículos ou de caixeiros viajantes, temos que levar em consideração as posições de cada vértice no grafo. Nestes casos, devemos criar um laço para obter as coordenadas x e y de cada vértice antes de construir os grafos. O exemplo mostrado a seguir chama-se pcb442 e trata-se de uma instância do problema clássico do caixeiro viajante (PCV ou TSP: traveling salesman problem).

```

import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

plt.figure()

position = np.array(pd.read_csv('C:/dados/pcb442.csv'))
df = pd.read_csv('C:/dados/pcb442_opt.csv')

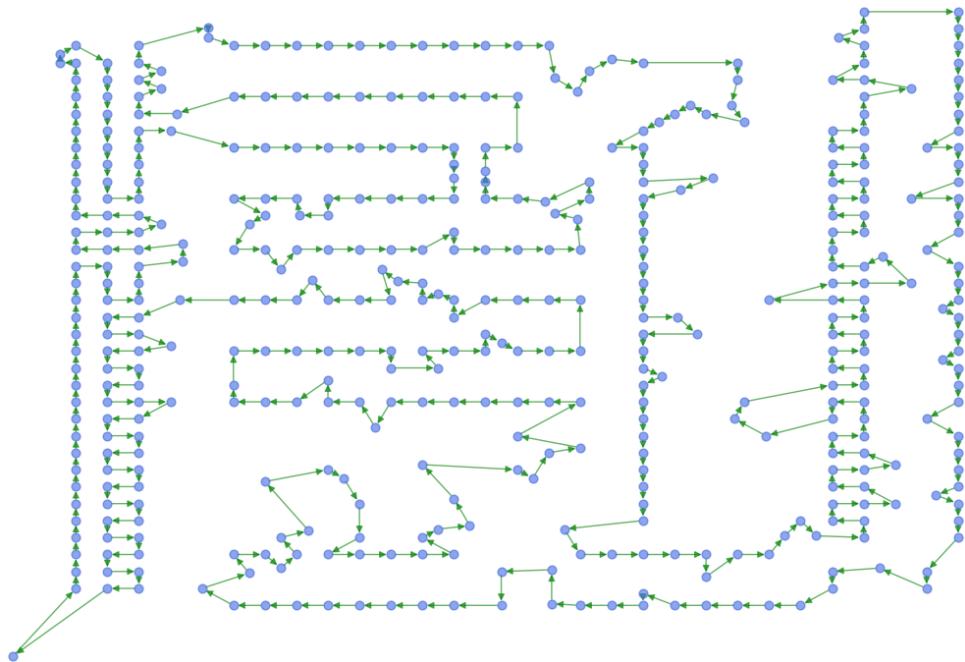
i = np.arange(0, len(df))
g = nx.from_pandas_edgelist(df, source = 'v1', target = 'v2', create_using = nx.DiGraph)
pos = {}

for k in i:
    pos[k] = [position[k][1], position[k][2]]

nx.draw(g, pos = pos, node_color = 'royalblue', edge_color = 'green', alpha = 0.6,
        linewidths = 1, node_size = 40)

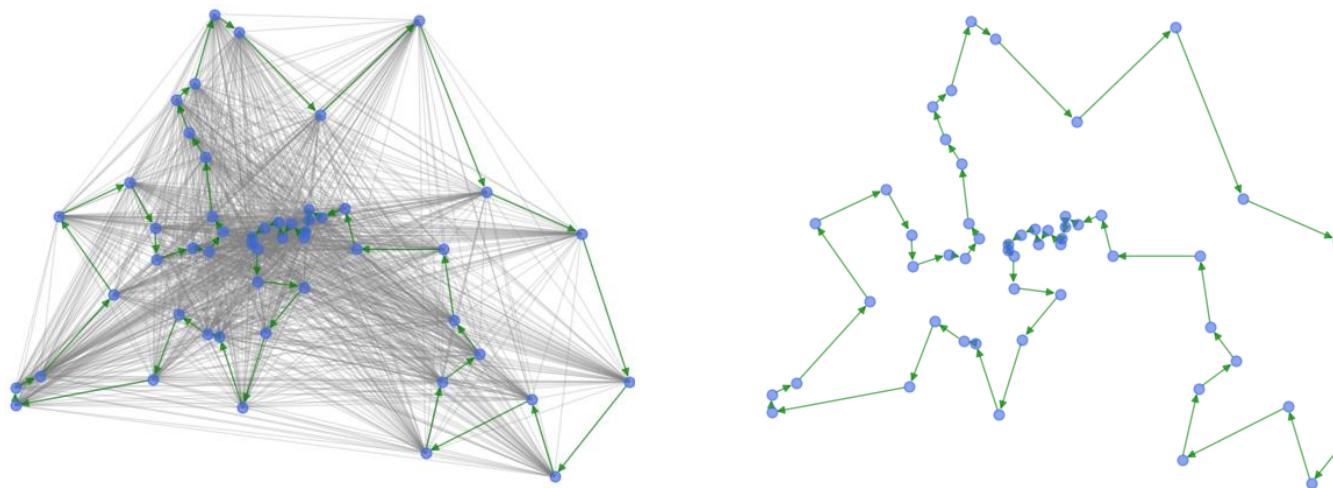
```

```
plt.show()
```



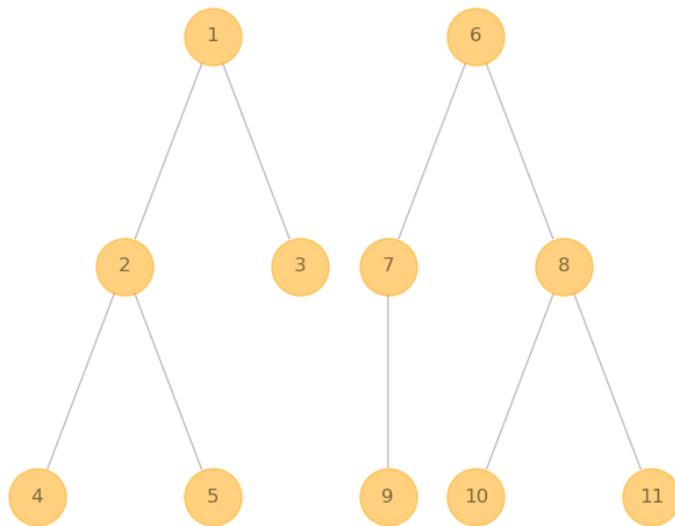
Dados disponíveis em: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/>

Nas representações de soluções para estes problemas com grafos, podemos programar uma interação com o usuário para mostrar todas as possíveis ligações entre os nós, e apenas a solução encontrada. No exemplo berlin52 mostrado a seguir, podemos perceber que uma visualização completa das ligações pode ser mostrada para que o usuário tenha a noção de diferentes soluções que podem ser encontradas neste tipo de optimização representado com grafos.



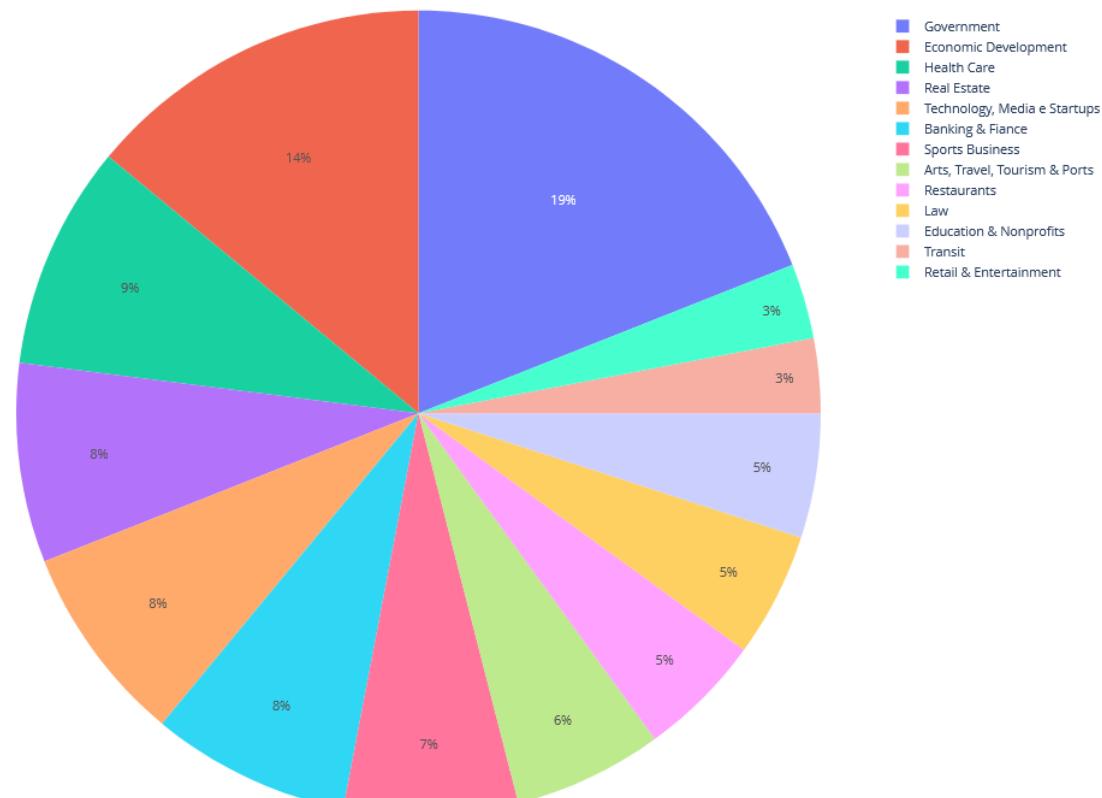
Dados disponíveis em: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/>

Em alguns casos, temos um conjunto de dados com informações de hierarquia entre os atributos, criando uma visualização do tipo árvore. Trata-se de um tipo especial de grafo sem ciclos, que pode ser representado por meio de uma técnica chamada de **TreeMap**.



Na técnica Treemap, temos um espaço retangular sendo dividido recursivamente em fatias, alterando entre horizontais e verticais, de acordo com as sub-árvore que representam o relacionamento entre os dados. Quando todo o conjunto de dados pertence a uma categoria, podemos utilizar os Treemaps como alternativa para os gráficos de setores circulares. No exemplo mostrado a seguir, temos os atributos de dados com seus respectivos valores que podem ser representados com um gráfico de setores com a função **pie** da biblioteca **plotly**:

<i>Government</i>	<i>Real Estate</i>	<i>Technology, Media e Startups</i>	<i>Banking & Fiance</i>	<i>Economic Development</i>
19	8	8	8	14
<i>Health Care</i>	<i>Sports Business</i>	<i>Arts, Travel, Tourism & Ports</i>	<i>Restaurants</i>	<i>Law</i>
9	7	6	5	5
<i>Education & Nonprofits</i>	<i>Retail & Entertainment</i>		<i>Transit</i>	
5	3		3	3



Dados disponíveis em: <https://infogram.com/3b2d4004-0a5c-49e9-ae88-5fdaaec5ff73>

```

import plotly.io as pio
import plotly.express as px
pio.renderers
pio.renderers.default = 'browser'

setores = ['Government', 'Real Estate', 'Technology, Media e Startups', 'Banking & Fiance',
           'Economic Development', 'Health Care', 'Sports Business',
           'Arts, Travel, Tourism & Ports', 'Restaurants', 'Law', 'Transit',
           'Education & Nonprofits', 'Retail & Entertainment']

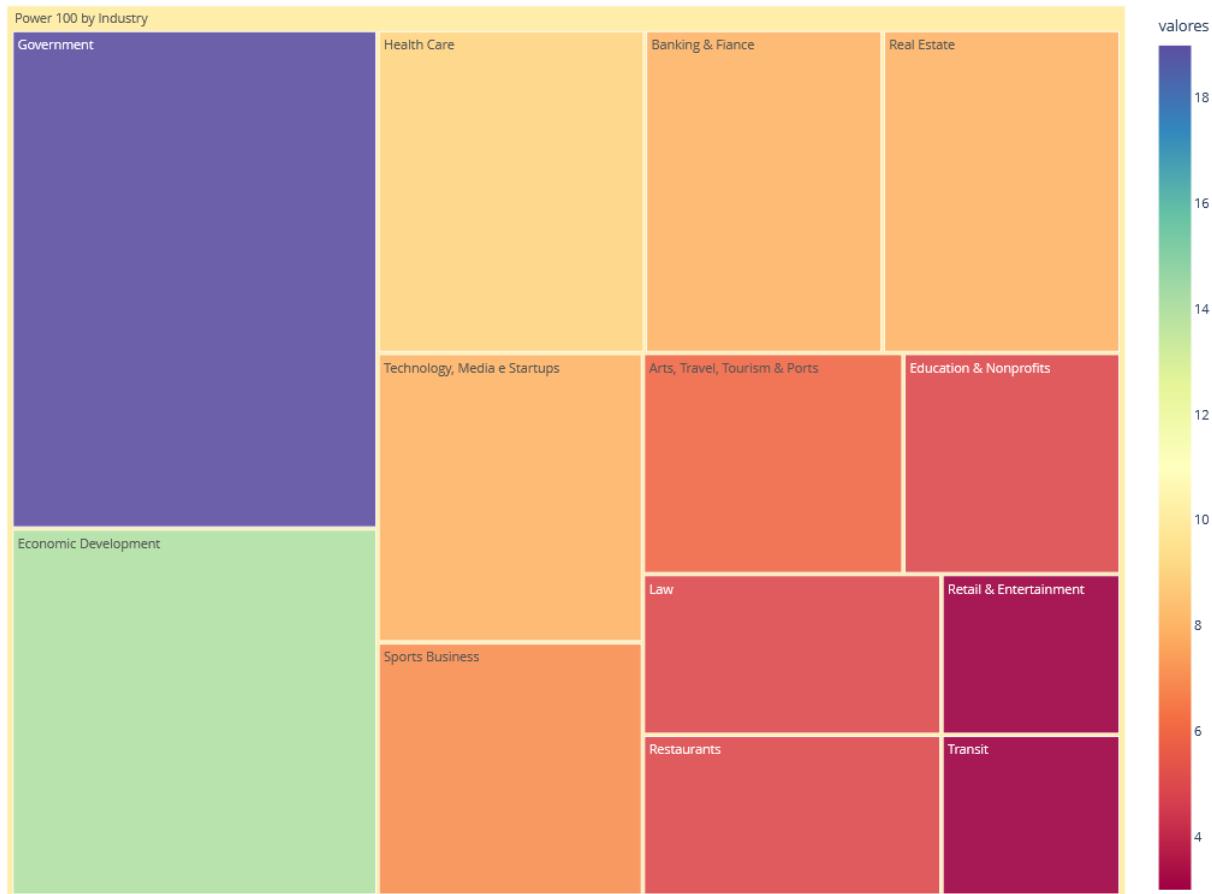
valores = [19, 8, 8, 8, 14, 9, 7, 6, 5, 5, 3, 5, 3]

fig = px.pie(values = valores, names = setores, opacity = 0.9)

fig.show()

```

Com a função DataFrame da biblioteca **pandas**, podemos criar um banco de dados com os setores e valores para utilizarmos na função treemap da biblioteca **plotly**. Desta forma, temos os dados de uma categoria representados com a técnica **Treeview**.



Dados disponíveis em: <https://infogram.com/3b2d4004-0a5c-49e9-ae88-5fdaaeecc5ff73>

```

import plotly.express as px
import pandas as pd
import plotly.io as pio
pio.renderers
pio.renderers.default = 'browser'

setores = ['Government', 'Real Estate', 'Technology, Media e Startups', 'Banking & Fiance',
           'Economic Development', 'Health Care', 'Sports Business',
           'Arts, Travel, Tourism & Ports', 'Restaurants', 'Law', 'Transit',
           'Education & Nonprofits', 'Retail & Entertainment']

```

```

valores = [19, 8, 8, 8, 14, 9, 7, 6, 5, 5, 3, 5, 3]

df = pd.DataFrame(dict(setores = setores, valores = valores))

df['Power 100 by Industry'] = 'Power 100 by Industry'

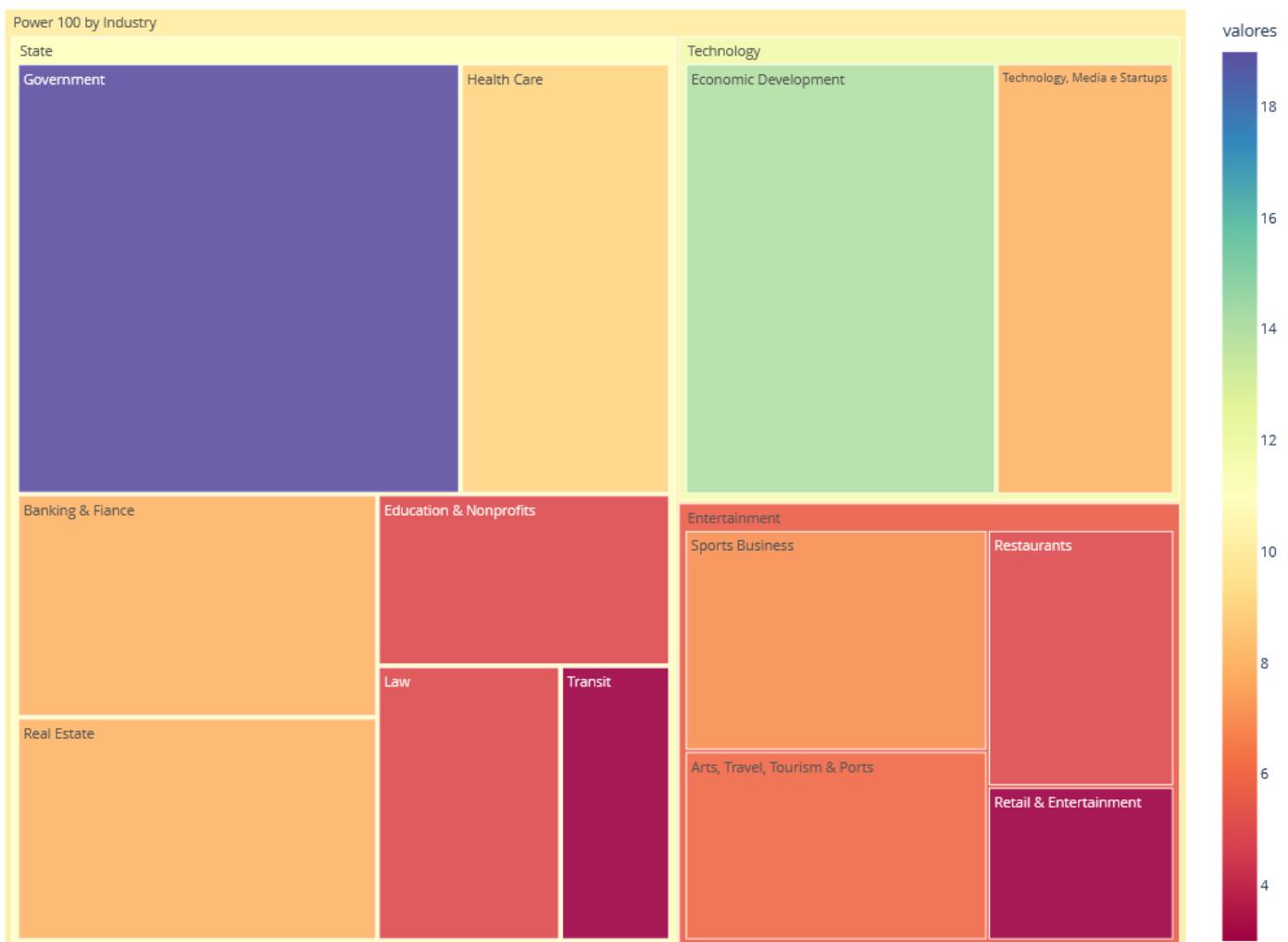
fig = px.treemap(df, path = ['Power 100 by Industry', 'setores'], values = 'valores',
                  color_continuous_scale = 'spectral', color = 'valores')

fig.update_traces(root_color = 'lightgrey', opacity = 0.9)
fig.update_layout(margin = dict(t = 25, l = 25, r = 25, b = 25))

fig.show()

```

Se criarmos as categorias **State**, **Technology** e **Entertainment**, podemos visualizar os dados agrupados com a técnica Treeview.



Dados disponíveis em: <https://infogram.com/3b2d4004-0a5c-49e9-ae88-5fdaeec5ff73>

```

import plotly.express as px
import pandas as pd
import plotly.io as pio
pio.renderers
pio.renderers.default = 'browser'

setores = ['Government', 'Real Estate', 'Technology, Media e Startups', 'Banking & Fiance',
          'Economic Development', 'Health Care', 'Sports Business',
          'Arts, Travel, Tourism & Ports', 'Restaurants', 'Law', 'Transit',
          'Education & Nonprofits', 'Retail & Entertainment']

```

```

valores = [19, 8, 8, 8, 14, 9, 7, 6, 5, 5, 3, 5, 3]

categorias = ['State', 'State', 'Technology', 'State', 'Technology', 'State', 'Entertainment',
    'Entertainment', 'Entertainment', 'State', 'State', 'State', 'Entertainment']

df = pd.DataFrame(dict(setores = setores, valores = valores, categorias = categorias))

df['Power 100 by Industry'] = 'Power 100 by Industry'

fig = px.treemap(df, path = ['Power 100 by Industry', 'categorias', 'setores'],
    values = 'valores', color_continuous_scale = 'spectral', color = 'valores')

fig.update_traces(root_color = 'lightgrey', opacity = 0.9)
fig.update_layout(margin = dict(t = 25, l = 25, r = 25, b = 25))

fig.show()

```

Com um banco de dados maior, podemos criar a visualização com o Treeview com a mesma biblioteca plotly, com a leitura de dados com a função da biblioteca pandas. O conjunto de dados mostrado a seguir contém informações para 100 grandes empresas de vários países, com dados de pesquisa e desenvolvimento de 2005 e 2006.



Dados disponíveis em: <https://www.treemap.com/datasets/rdspenders/>

```

import plotly.express as px
import plotly.io as pio
import pandas as pd
import numpy as np
pio.renderers
pio.renderers.default = 'browser'

```

```
df = pd.read_csv('C:/dados/treemap1.csv')

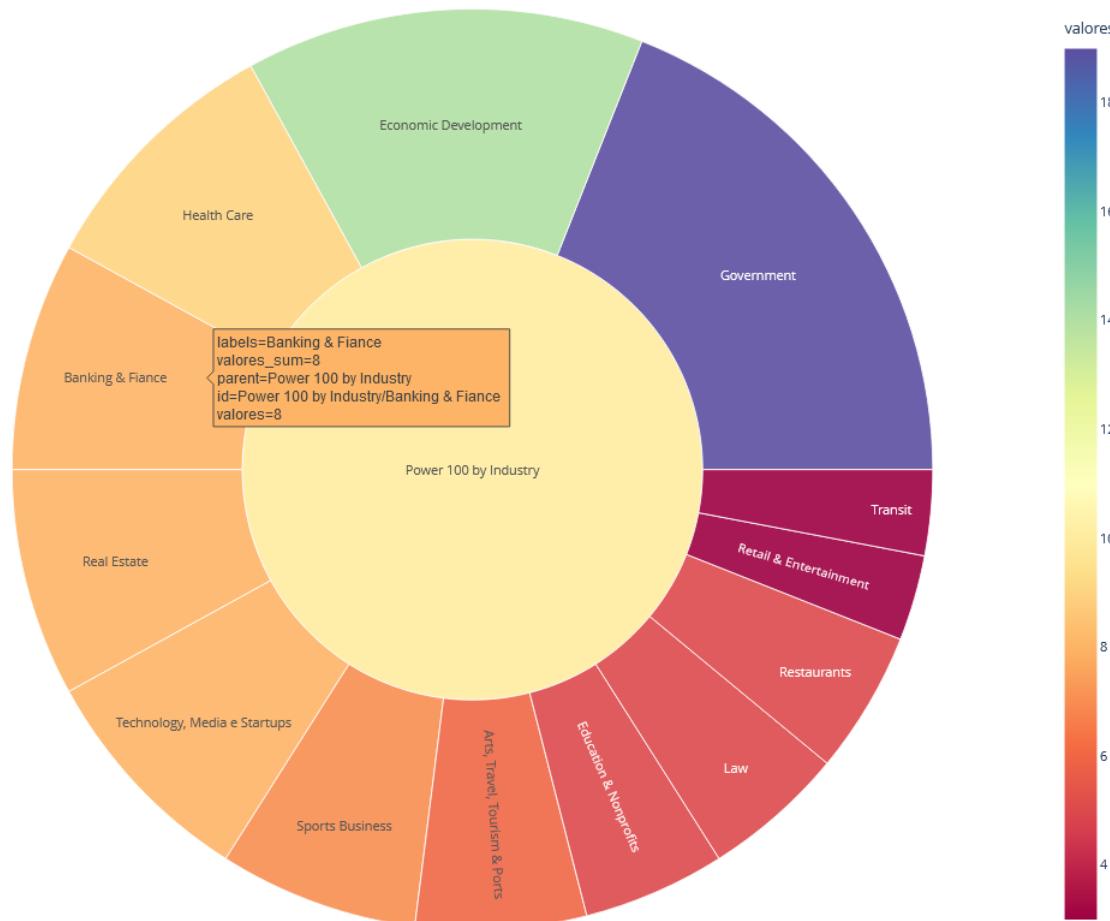
fig = px.treemap(df, path = [px.Constant('IEEE Spectrum'), 'Country', 'Company'],
                  values = 'Sales 2005', color = 'Sales 2005', hover_data = ['Sales 2006'],
                  color_continuous_scale = 'rainbow',
                  color_continuous_midpoint = np.average(df['Sales 2006'], weights = df['Rank 2006']))

fig.update_layout(margin = dict(t = 25, l = 25, r = 25, b = 25))

fig.show()
```

Outra técnica hierárquica conhecida é chamada **Sunburst**. Nesta técnica, a partir do centro da representação são colocados os agrupamentos por meio de anéis aninhados, indicando as respectivas camadas da hierarquia. A estrutura do código para uma visualização Sunburst é a mesma da Treeview. Portanto, podemos usar os mesmos códigos dos exemplos de Treeview para representar as visualizações com a técnica Sunburst para os exemplos apresentados anteriormente. No primeiro exemplo, podemos substituir apenas a linha de definição da visualização:

```
fig = px.sunburst(df, path = ['Power 100 by Industry', 'setores'], values = 'valores',
                   color_continuous_scale = 'spectral', color = 'valores')
```

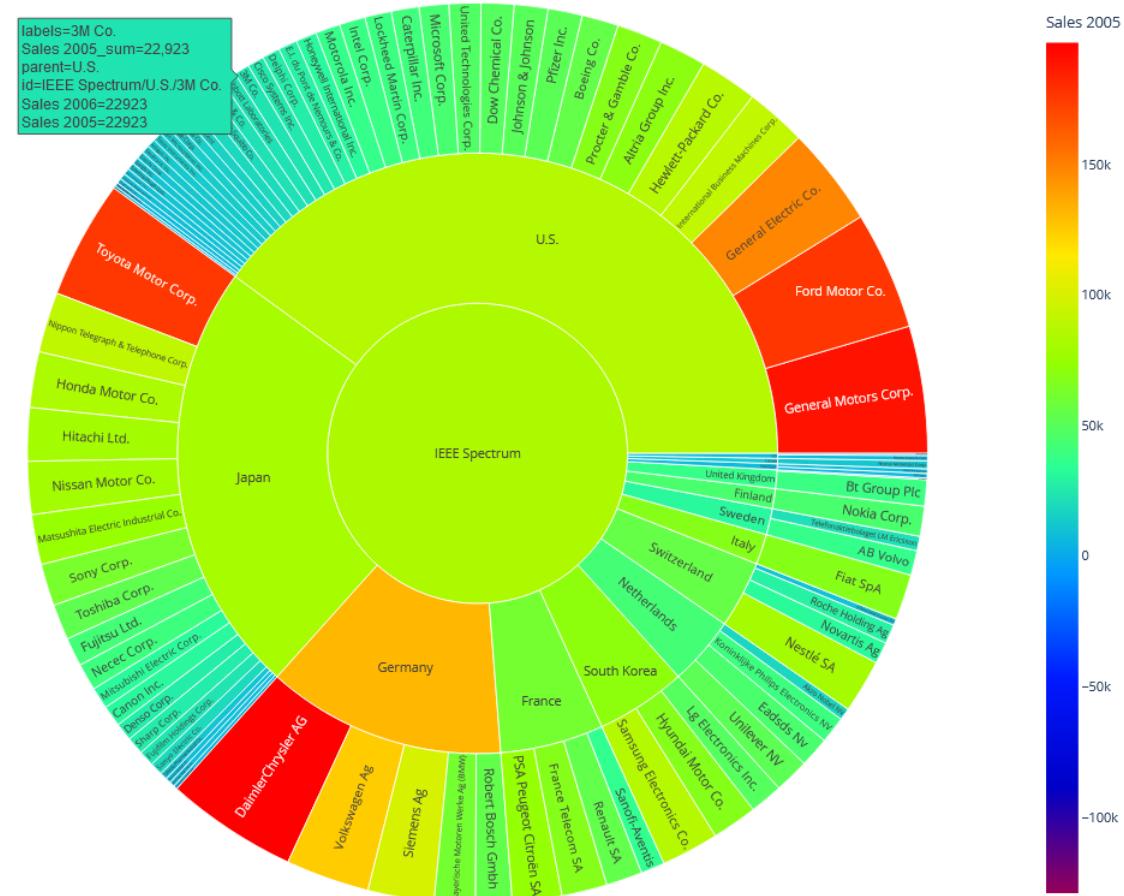
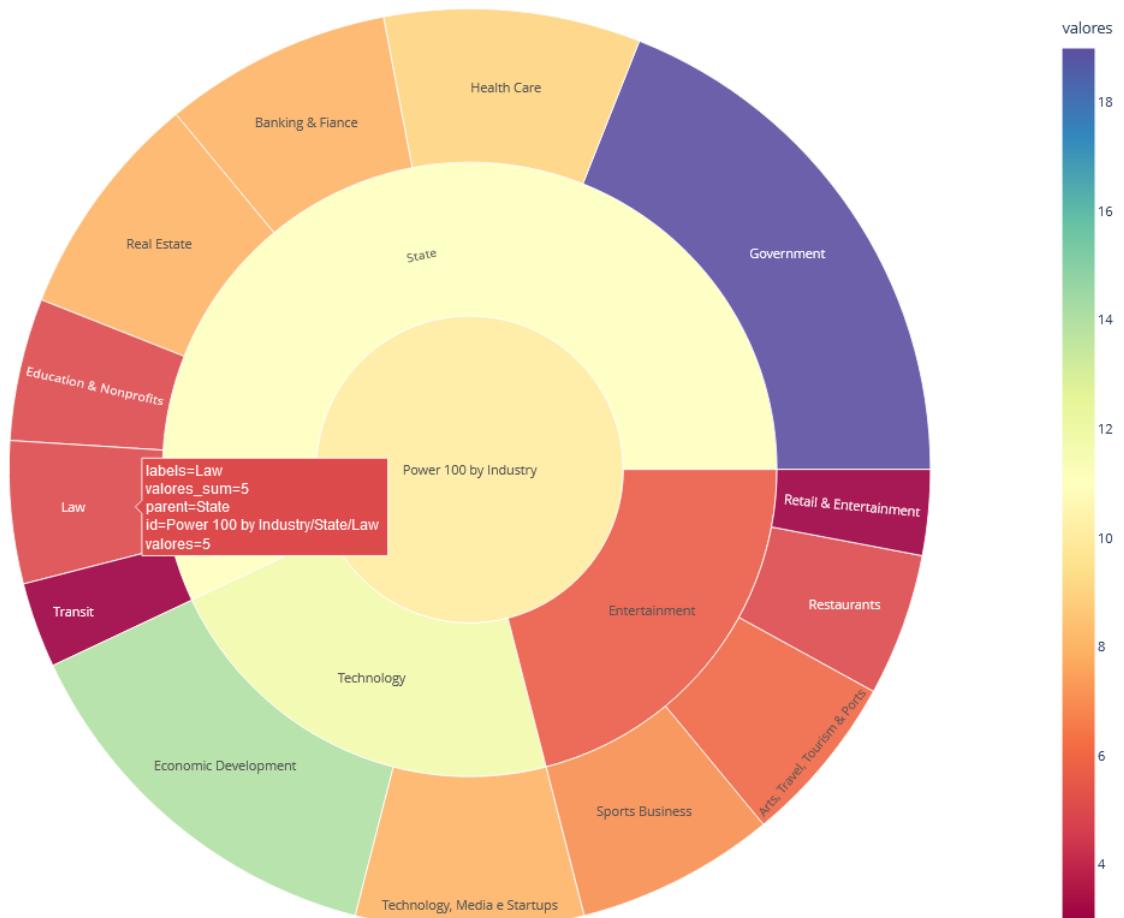


No segundo exemplo, a linha de definição da visualização pode ser escrita da seguinte forma:

```
fig = px.sunburst(df, path = ['Power 100 by Industry', 'categorias', 'setores'],
                   values = 'valores', color_continuous_scale = 'spectral', color = 'valores')
```

No terceiro exemplo, temos a linha da definição da visualização escrita da seguinte maneira:

```
fig = px.sunburst(df, path = [px.Constant('IEEE Spectrum'), 'Country', 'Company'],
                  values = 'Sales 2005', color = 'Sales 2005', hover_data = ['Sales 2006'],
                  color_continuous_scale = 'rainbow',
                  color_continuous_midpoint = np.average(df['Sales 2006'], weights = df['Rank 2006']))
```



O Radviz é um algoritmo de visualização de dados multivariados que plota cada dimensão de atributo do conjunto de dados uniformemente, ao redor de uma circunferência de raio unitário centrada na origem do sistema

de coordenadas. Os pontos no interior do círculo são plotados normalizando os valores de suas coordenadas nos eixos do centro para cada eixo de atributos.

Este mecanismo permite tantas dimensões quantas cabem facilmente em um círculo, aumentando consideravelmente a dimensionalidade da visualização. Este método pode ser usado para detectarmos a separabilidade entre as classes. Cada atributo determina uma espécie de âncora $A_j = (Ax_j, Ay_j)$ em torno da circunferência de raio unitário, e as coordenadas dos dados são determinadas com uma espécie de média ponderada entre valores dos atributos:

$$x_p = \frac{\sum_{j=0}^{n-1} d_{p,j} \cdot Ax_j}{\sum_{j=0}^{n-1} d_{p,j}}, \quad y_p = \frac{\sum_{j=0}^{n-1} d_{p,j} \cdot Ay_j}{\sum_{j=0}^{n-1} d_{p,j}}.$$

Considere o exemplo com 4 atributos com as âncoras A_1, A_2, A_3 e A_4 que determinam os vetores:

$$Ax = [1, 0, -1, 0] \text{ e } Ay = [0, 1, 0, -1].$$

<i>id</i>	Atributo 1	Atributo 2	Atributo 3	Atributo 4
1	5	2	1	2
2	7	5	1	7
3	2	4	1	3

Para o dado 1, temos as seguintes coordenadas:

$$x_1 = \frac{5 \cdot 1 + 2 \cdot 0 + 1 \cdot (-1) + 2 \cdot 0}{5+2+1+2} = \frac{4}{10} = 0,4$$

$$y_1 = \frac{5 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1)}{5+2+1+2} = \frac{0}{10} = 0$$

Para o dado 2, temos as seguintes coordenadas:

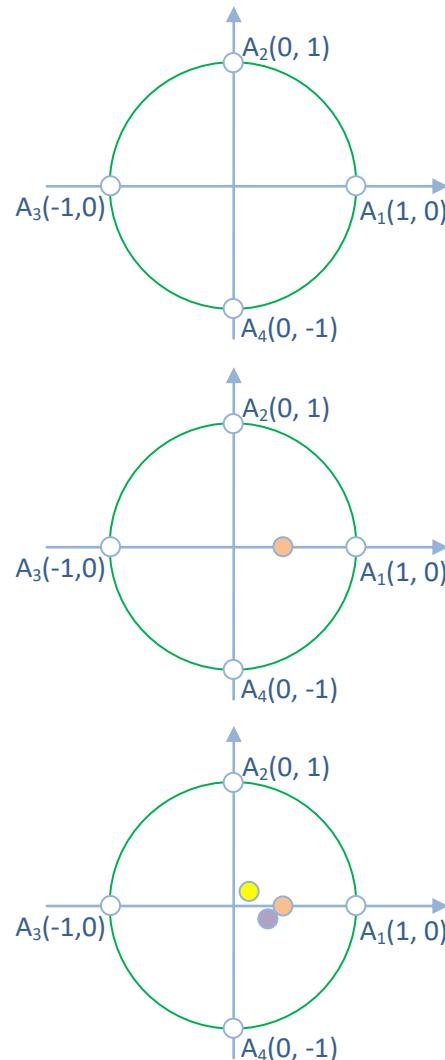
$$x_1 = \frac{7 \cdot 1 + 5 \cdot 0 + 1 \cdot (-1) + 7 \cdot 0}{7+5+1+7} = \frac{6}{20} = 0,3$$

$$y_1 = \frac{7 \cdot 0 + 5 \cdot 1 + 1 \cdot 0 + 7 \cdot (-1)}{7+5+1+7} = \frac{-2}{20} = -0,1$$

Para o dado 3, temos as seguintes coordenadas:

$$x_1 = \frac{2 \cdot 1 + 4 \cdot 0 + 1 \cdot (-1) + 3 \cdot 0}{2+4+1+3} = \frac{1}{10} = 0,1$$

$$y_1 = \frac{2 \cdot 0 + 4 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)}{2+4+1+3} = \frac{1}{10} = 0,1$$



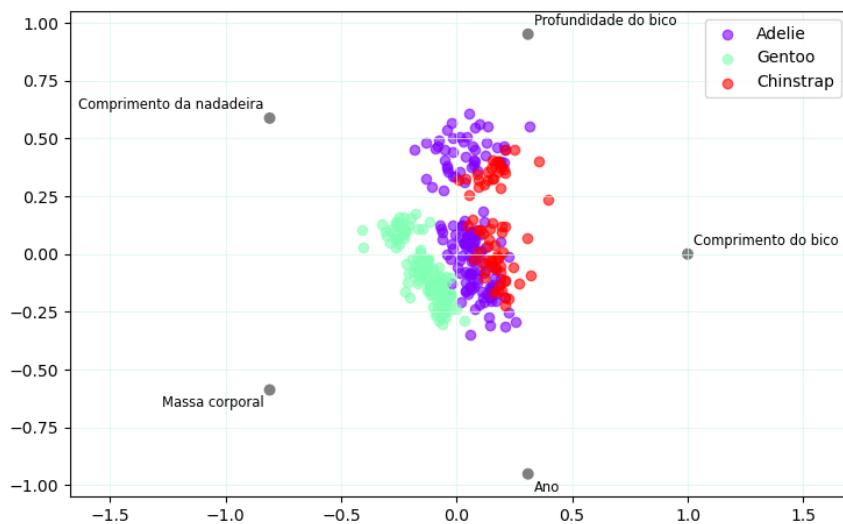
Com o exemplo dos dados do conjunto dos pinguins, temos a representação por meio da técnica **RadViz** com 5 atributos com o uso da biblioteca **pandas**. Devemos selecionar as colunas para evitar que informações com textos sejam usadas para os cálculos, além de não considerar a coluna de identificadores dos dados (geralmente é a coluna 0).

```
import pandas as pd
from matplotlib import pyplot as plt

pinguin = pd.read_csv('C:/dados/penguin2.csv', header = 0, usecols = [1,3,4,5,6,8])

ax = plt.grid(color = '#d5f8e3', linewidth = 0.5)
```

```
fig = pd.plotting.radviz(pinguin, 'Espécie', colormap = 'rainbow', alpha = 0.6, ax = ax)
fig.show
```

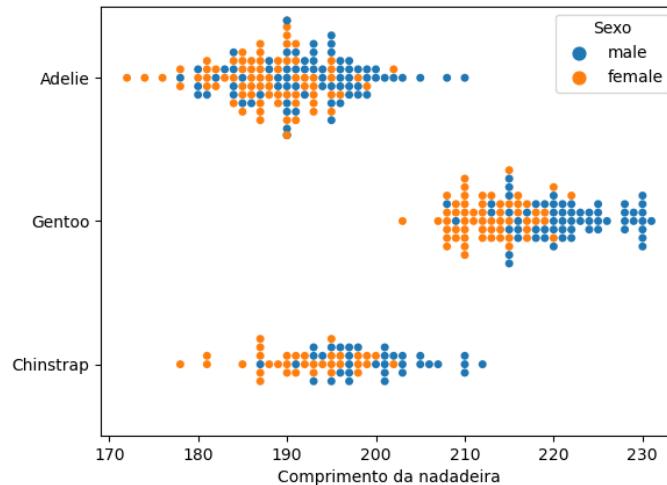


O gráfico de enxame (swarm) consiste em outra maneira de representar a distribuição de um atributo ou a distribuição conjunta de alguns atributos. Um dos eixos representa um atributo escolhido, enquanto o outro eixo mostra os atributos separados com um critério.

No exemplo do conjunto dos pinguins, podemos escolher no eixo x um atributo, e no eixo y as espécies. Além disso, podemos usar as cores como critério de separação de outro atributo.

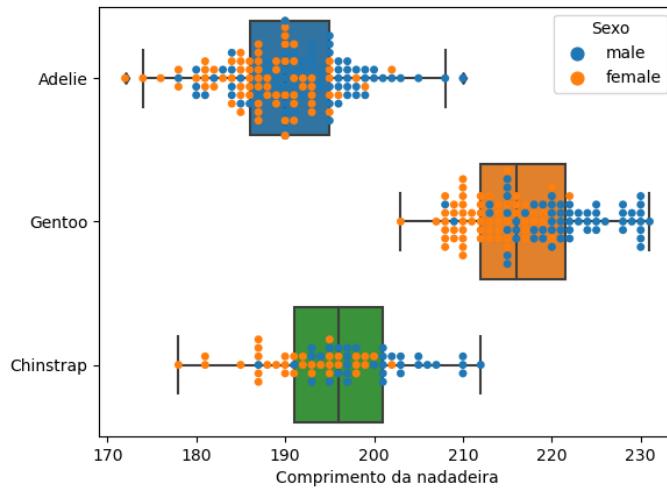
```
import pandas as pd
import seaborn as sns

pinguin = pd.read_csv('C:/dados/penguin2.csv')
sns.swarmplot(x = 'Comprimento da nadadeira', y = 'Espécie', hue = 'Sexo', data = pinguin)
```



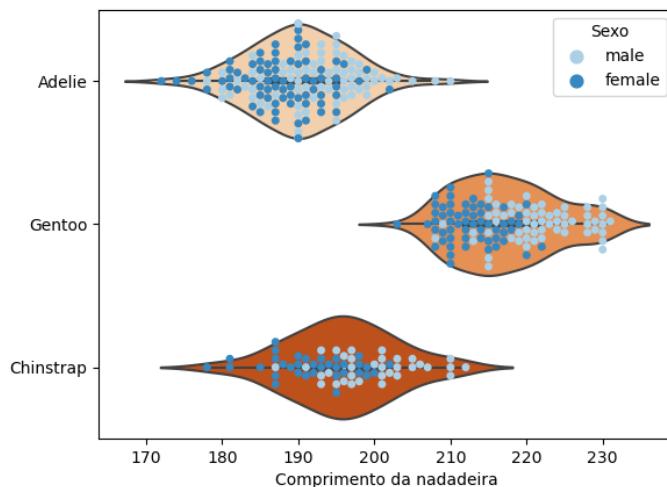
Os dados com este tipo de representação por enxame podem ser mostrados junto com os diagramas de caixas (boxplots), que mostram a variação de dados observados de cada variável numérica por meio de quartis.

```
sns.boxplot(x = 'Comprimento da nadadeira', y = 'Espécie', data = pinguin)
sns.swarmplot(x = 'Comprimento da nadadeira', y = 'Espécie', hue = 'Sexo', data = pinguin)
```



Outro gráfico que pode ser mostrado em conjunto com a representação por enxame é chamado de violino (violinplot). Esta representação é parecida com o boxplot, porém é feita utilizando a densidade de probabilidade dos dados por meio do núcleo de distribuição de cada variável. A representação da curva que lembra um violino tem a suavização por um estimador de densidade do núcleo de dados de cada variável.

```
sns.violinplot(x = 'Comprimento da nadadeira', y = 'Espécie', data = pinguin,
                palette = 'Oranges')
sns.swarmplot(x = 'Comprimento da nadadeira', y = 'Espécie', hue = 'Sexo', data = pinguin,
               palette = 'Blues')
```



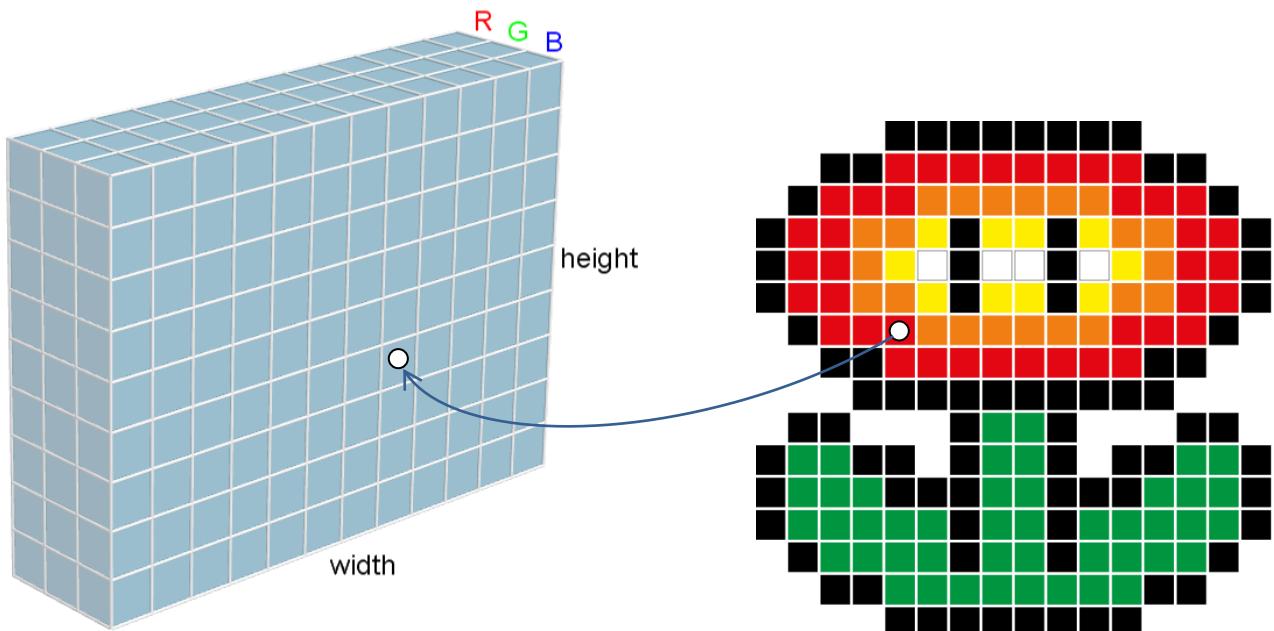
4.2. Reconhecimento de imagens

O reconhecimento de imagem refere-se à tarefa de inserir uma imagem como entrada para uma técnica de classificação de padrões (geralmente utilizamos uma rede neural artificial), produzindo algum tipo de rótulo para essa imagem. O rótulo que a técnica determina corresponde a uma classe pré-definida. Podemos ter várias classes com as quais a imagem pode ser rotulada ou apenas uma. Se existir uma única classe para cada imagem, o termo "reconhecimento" pode ser aplicado, enquanto uma tarefa de reconhecimento de várias classes pode ser chamada de "classificação".

Um subconjunto de classificação de imagem é a detecção de objetos, onde instâncias específicas de objetos são identificadas como pertencentes a uma determinada classe, como carros, animais, paisagens, pessoas ou imagens para diagnósticos médicos. Para realizar o reconhecimento/classificação de imagens, devemos realizar um pré-processamento de cada imagem chamado de extração de características. Os atributos dos objetos devem ser escolhidos para formar o conjunto de entrada da técnica de classificação de padrões. No caso específico de reconhecimento de imagem, as características são os grupos de pixels, arestas e pontos de objetos que a técnica analisará em busca de padrões.

O reconhecimento de recursos (ou extração) é o processo de extrair as informações relevantes de uma imagem de entrada para que essas informações possam ser analisadas. Muitas imagens contêm anotações ou metadados sobre a imagem que ajudam as técnicas de reconhecimento de padrões a encontrar as combinações de atributos relevantes.

Uma imagem pode ser vista como uma matriz de pixels, que pode ser dividida em três canais de cores: vermelho (R), verde (G) e azul (B). As informações contidas nas imagens não estão estruturadas neste formato matricial, e podemos extrair estas informações com o uso da biblioteca **pillow** com o código mostrado a seguir. Os dados podem ser conferidos em um arquivo salvo em formato CSV.

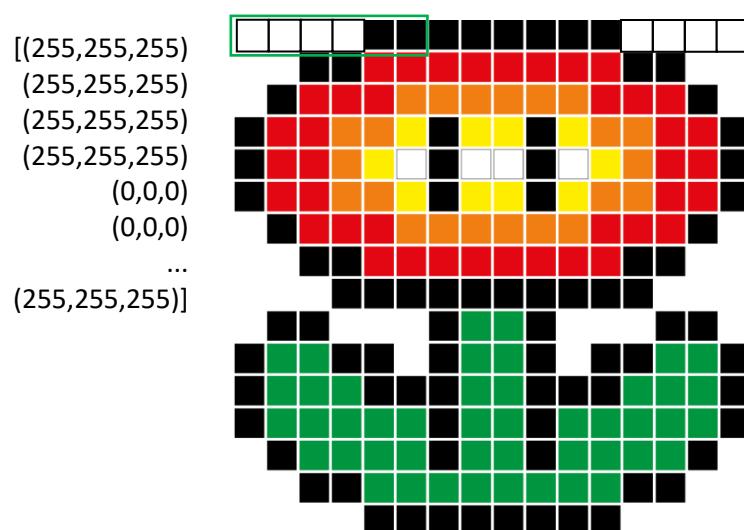


```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

im = Image.open('C:/dados/imagem.png').convert('RGB')

imf = np.asarray(im)
imf.tofile('C:/dados/testeImagem.csv', sep = ',')

print(im.size)
plt.imshow(imf)
print(imf)
```



Com os dados da matriz de canais RGB de cada imagem, podemos extrair características estatísticas ou de posições de arestas e pontos específicos. Para calcular média e desvio padrão de cada cor, podemos usar as seguintes funções:

```
Rm = np.mean(imf[:,1])
Gm = np.mean(imf[:,2])
Bm = np.mean(imf[:,3])
Rd = np.std(imf[:,1])
Gd = np.std(imf[:,2])
Bd = np.std(imf[:,3])
```

Outras funções disponíveis nesta biblioteca são relativas à edição de imagem, tais como: cortes, filtros, redimensionamento e reposicionamento. As funções de espelhamento são interessantes e têm os seguintes valores disponíveis: FLIP_LEFT_RIGHT, FLIP_TOP_BOTTOM, ROTATE_90, ROTATE_180, ROTATE_270 e TRANSPOSE.

```
im = im.transpose(Image.FLIP_LEFT_RIGHT)
```



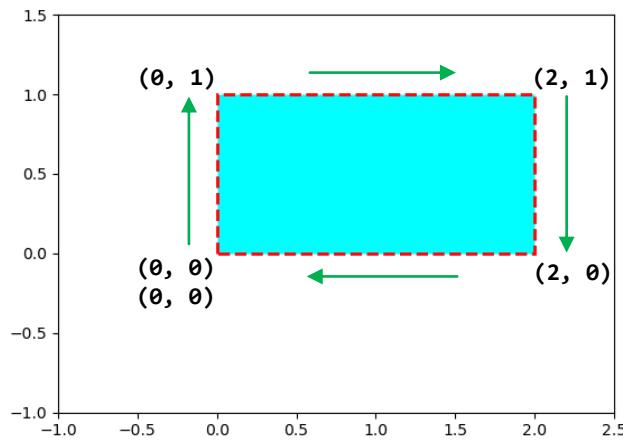
Atividade 4

- 4.1. Escolha um conjunto de dados, que contém pelo menos 3 variáveis, para representá-lo por meio de gráficos do tipo radar ou grafo.
- 4.2. Utilize a biblioteca bokeh para fazer as seleções interativas de dados de classificação de padrões, colocados em pelo menos 2 gráficos diferentes.
- 4.3. Escolha um conjunto de dados para fazer a visualização em formato TreeView e Sunburst.
- 4.4. Escolha um conjunto de imagens para fazer a extração de características estatísticas sobre suas respectivas matrizes de pixels. Faça uma análise dos dados destas matrizes, fazendo uma pré-classificação das mesmas.

5. Linhas, polígonos, poliedros e superfícies

Agora vamos estudar as representações de poligonais fechadas em 2D. A ideia básica é a mesma que utilizamos em linguagens como Java e SVG: criamos a sequência de linhas que percorre cada vértice do polígono e depois de percorrer todos os vértices definimos o fechamento da poligonal e os atributos de preenchimento (cor, transparência e gradiente) e os atributos de linha (contínua ou tracejada e cor).

No exemplo mostrado a seguir, temos a construção de um retângulo com lados de medidas 2 e 1 com as coordenadas indicadas em `verts`. O comando `Path` desenha o retângulo na sequência do conjunto de vértices `verts`.



```

import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [(0, 0), (0, 1), (2, 1), (2, 0), (0, 0),]
path = Path(verts)

fig, ax = plt.subplots()
patch = patches.PathPatch(path, facecolor = 'aqua', linestyle = '--', linewidth = 2,
                           edgecolor = 'red')
ax.add_patch(patch)

ax.set_xlim(-1, 2.5)
ax.set_ylim(-1, 1.5)
plt.gca().set_aspect('equal', adjustable = 'box')

plt.show()

```

Usando o comando `ax.text`, podemos adicionar rótulos para os vértices representados.

```

ax.text(0, 0, 'A')
ax.text(0, 1, 'B')
ax.text(2, 1, 'C')
ax.text(2, 0, 'D')

```

Construa um triângulo e um pentágono usando os comandos de poligonais. Outras figuras que podemos representar são círculos e elipses. Basta usar as funções destas figuras com a referência da biblioteca indicada. Veja o exemplo do código para representar uma elipse de diâmetros 0.7 e 0.3, com centro no ponto (0.5, 0.5):

```

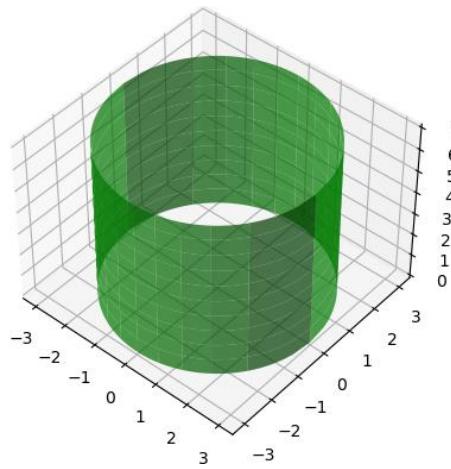
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

fig, ax = plt.subplots()
patch = Ellipse((0.5, 0.5), 0.7, 0.3, color = 'orange')
ax.add_patch(patch)

plt.show()

```

Construa um círculo usando a referência de biblioteca `Circle`. Agora vamos representar superfícies 3D usando polígonos, elipses, círculos ou suas respectivas equações. Um cilindro circular reto com altura 4 e raio 3 pode ser representado por meio do código mostrado a seguir. Utilizamos a função `meshgrid` para criar a superfície definida em 2 dimensões (neste caso, x e z), e a equação que define os círculos das bases do cilindro está definida na coordenada y . A superfície fica definida com linhas auxiliares, como uma malha, com as medidas `rstride` (linha) e `cstride` (coluna).



```

import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

raio = 3
altura = 7
x = np.linspace(-raio, raio, 100)
z = np.linspace(0, altura, 100)
x1, z1 = np.meshgrid(x, z)
y1 = np.sqrt(raio**2-x1**2)

rstride = 10
cstride = 10
ax.plot_surface(x1, y1, z1, alpha = 0.7, color = 'green', rstride = rstride, cstride = cstride)
ax.plot_surface(x1, -y1, z1, alpha = 0.7, color = 'green', rstride = rstride, cstride= cstride)

plt.show()

```

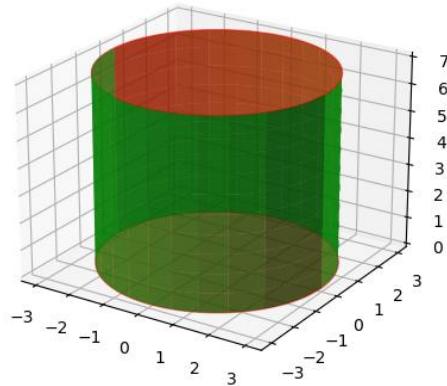
Note que temos somente a representação da superfície lateral do cilindro. Os círculos das bases podem ser definidos com a referência patches de círculos e a função de representação de elementos 2D para 3D:

```

from matplotlib.patches import Circle
import mpl_toolkits.mplot3d.art3d as art3d

p = Circle((0, 0), raio, color = 'red', alpha = 0.5)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z = 0, zdir = 'z')
p = Circle((0, 0), raio, color = 'red', alpha = 0.5)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z = altura, zdir = 'z')

```



Para representar um cone circular reto, podemos usar a mesma ideia do cilindro.

```

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Circle
import mpl_toolkits.mplot3d.art3d as art3d

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

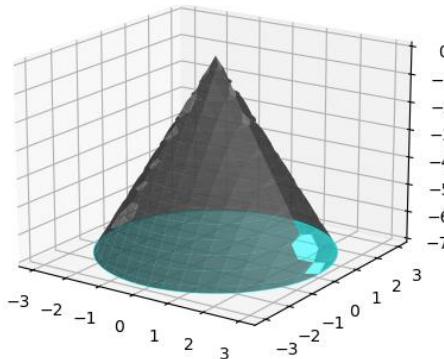
raio = 3
altura = 7
x = np.linspace(-raio, raio, 155)
z = np.linspace(0, altura, 155)
x1, z1=np.meshgrid(x, z)
y1 = np.sqrt(z1**2*(raio/altura)**2 - x1**2)

rstride = 10
cstride = 10
ax.plot_surface(x1, y1, -z1, alpha = 0.7, color = 'grey', rstride = rstride, cstride = cstride)
ax.plot_surface(x1, -y1, -z1, alpha = 0.7, color = 'grey', rstride = rstride, cstride= cstride)

p = Circle((0, 0), raio, color = 'aqua', alpha = 0.5)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z = -altura, zdir = 'z')

plt.show()

```



Por causa do uso da raiz quadrada, alguns pontos ficam definidos com erros. Por isso temos alguns “furos” nesta representação gráfica. Utilizando as coordenadas polares, conseguimos uma representação perfeita do cone.

```

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

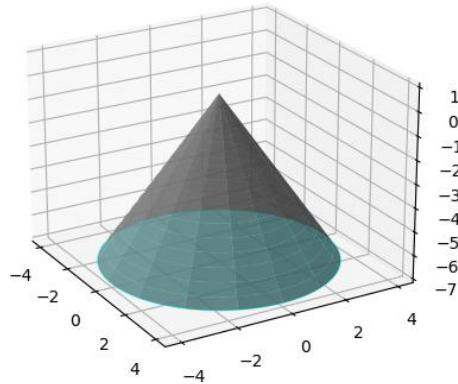
raio = 4
altura = 7
altura1 = np.linspace(0, altura, 150)
raio1 = np.linspace(0, raio, 150)
theta = np.linspace(0, 2*np.pi, 150)

R, T = np.meshgrid(raio1, theta)
A, T = np.meshgrid(altura1, theta)
X, Y, Z = R*np.cos(T), R*np.sin(T), A

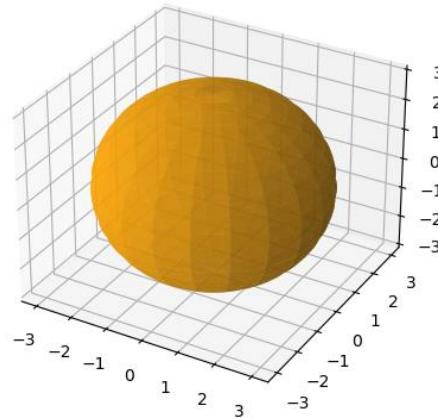
p = Circle((0, 0), raio, color = 'aqua', alpha = 0.2)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z = -altura, zdir = 'z')

rstride = 10
cstride = 10
ax.plot_surface(X, Y, -Z, alpha = 0.7, color = 'grey', rstride = rstride, cstride = cstride)

```

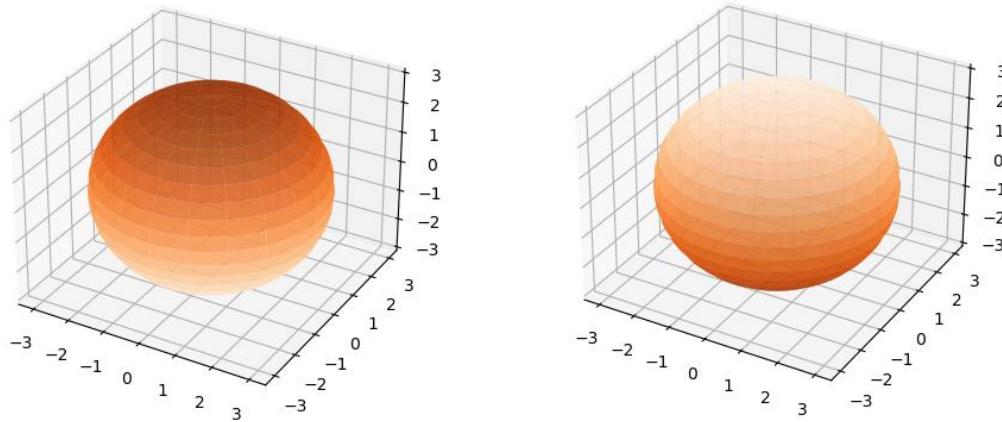


Agora modifique a estrutura do código de coordenadas do cilindro para coordenadas polares. Crie o código para representar uma esfera de raio qualquer.



O comando de superfícies permite o uso dos Mapas de Cores, que podem ser escolhidos para melhorar a representação de um objeto ou de um conjunto de dados em 3D. Usando o mapa de cor Oranges, temos a primeira representação mostrada a seguir. Para inverter a direção de qualquer mapa de cor, basta colocar _r no final do respectivo nome. A segunda representação da esfera, mostrada a seguir, tem o mapa de cor invertido.

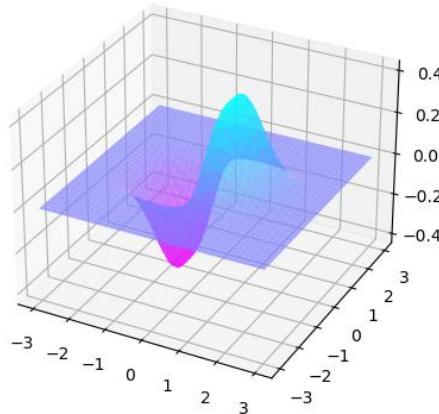
```
ax.plot_surface(X, Y, Z, alpha = 0.7, cmap = 'Oranges', rstride = rstride, cstride = cstride)
ax.plot_surface(X, Y, Z, alpha = 0.7, cmap = 'Oranges_r', rstride = rstride, cstride = cstride)
```



Usando a função `meshgrid` para as variáveis x e y, podemos criar outras superfícies por meio de códigos similares a este:

```
r = 3
x = np.linspace(-r, r, 155)
y = np.linspace(-r, r, 155)
x1, y1 = np.meshgrid(x, y)
z1 = x1*np.exp(-x1**2 - y1**4)
```

```
ax.plot_surface(x1, y1, z1, alpha = 0.7, cmap = 'cool_r')
```

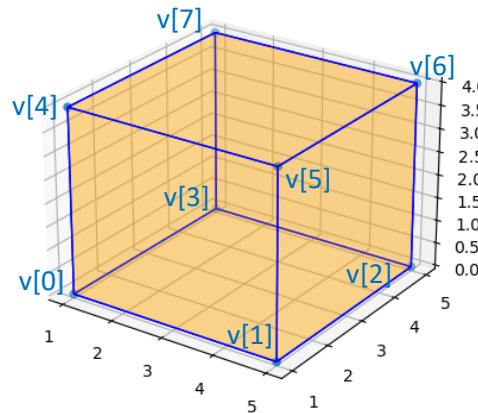


Agora vamos acompanhar a construção de poliedros usando os comandos de listas de vértices e a biblioteca **Poly3DCollection**. Começamos criando a lista de vértices que fica armazenada na matriz *v*. É interessante colocar os vértices em uma ordem que seja fácil de relacionar com as faces, pois faremos uma lista de vértices para as faces. Neste caso, o cubo tem aresta 4, com o ponto *v[0]* de coordenadas (1, 1, 0), o ponto *v[1]* com coordenadas (5, 1, 0) e assim por diante:

```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

v = np.array([[1, 1, 0], [5, 1, 0], [5, 5, 0], [1, 5, 0], [1, 1, 4], [5, 1, 4], [5, 5, 4],
              [1, 5, 4]])
```



A lista de faces contém os subconjuntos de vértices que formam cada face. Começando da face horizontal de coordenada *z* = 0, temos *faces[0]* = *v[0]*, *v[1]*, *v[2]*, *v[3]*. A face com coordenada *y* = 1 será *faces[1]* = *v[0]*, *v[5]*, *v[4]*, *v[3]*. Fazendo o mesmo com as outras 6 faces, temos a matriz indicadora de vértices:

```
faces = [[v[0],v[1],v[2],v[3]], [v[0],v[1],v[5],v[4]], [v[0],v[3],v[7],v[4]],
          [v[3],v[2],v[6],v[7]], [v[1],v[2],v[6],v[5]], [v[4],v[5],v[6],v[7]]]
```

Outra relação que deve ser feita é de quais coordenadas serão utilizadas para criar o poliedro. Neste caso, devemos usar a referência de que as coordenadas *x* serão as coordenadas da primeira posição de cada vértice, *y* serão as coordenadas da segunda posição e *z* da terceira posição.

```
ax.scatter3D(v[:, 0], v[:, 1], v[:, 2])
```

Escolhemos as cores de linhas, das faces e a opacidade como parâmetros dos comandos *add_collection3d* e *Poly3DCollection* para representar o poliedro.

```
ax.add_collection3d(Poly3DCollection(faces, facecolors = 'orange', edgecolors = 'blue',
alpha = 0.25))
```

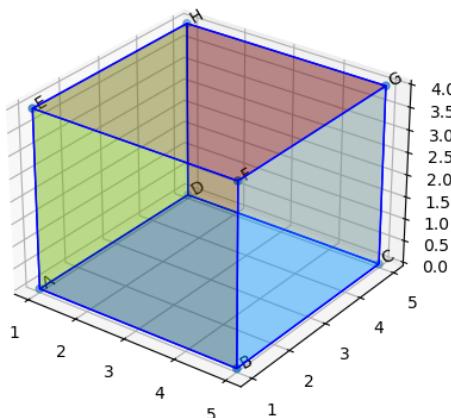
Podemos definir um vetor de cores para criar poliedros com cores diferentes em cada face. Como temos 8 faces, nosso vetor de cores tem 8 posições que serão atribuídas na ordem de definição que criamos na matriz faces.

```
cores = ['blue', 'green', 'yellow', 'red', 'cyan', 'black']
```

```
ax.add_collection3d(Poly3DCollection(faces, facecolors = cores, edgecolors = 'blue',
alpha = 0.25))
```

Outra lista que podemos criar está vinculada aos vértices do poliedro. Cada vértice pode ser melhor definido se usarmos uma lista de rótulos para a representação gráfica do poliedro. Primeiro devemos referenciar quais são as coordenadas x, y e z para usarmos os rótulos com o atributo **text3D**.

```
x = v[:, 0]
y = v[:, 1]
z = v[:, 2]
rotulos = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
for x, y, z, tag in zip(x, y, z, rotulos):
    label = tag
    ax.text3D(x, y, z, label, zdir = [1,1,1], color = 'k')
```



Usando os comandos apresentados para representação do cubo, faça a representação gráfica de uma pirâmide reta de base quadrada. Represente também uma pirâmide oblíqua de base quadrada. Nestes exemplos, podemos inserir o seguinte código para deixar os eixos na mesma escala.

```
def set_axes_equal(ax: plt.Axes):
    limits = np.array([ax.get_xlim3d(), ax.get_ylim3d(), ax.get_zlim3d()])
    origin = np.mean(limits, axis=1)
    radius = 0.5 * np.max(np.abs(limits[:, 1] - limits[:, 0]))
    _set_axes_radius(ax, origin, radius)

def _set_axes_radius(ax, origin, radius):
    x, y, z = origin
    ax.set_xlim3d([x - radius, x + radius])
    ax.set_ylim3d([y - radius, y + radius])
    ax.set_zlim3d([z - radius, z + radius])

set_axes_equal(ax)
```

Podemos definir os limites dos eixos usando os seguintes comandos:

```
ax.set_xlim(0, np.max(v[:, 0]))
ax.set_ylim(0, np.max(v[:, 1]))
ax.set_zlim(0, np.max(v[:, 2]))
```

Com estes valores definidos, os eixos x, y e z mostram os valores das três coordenadas começando no ponto de origem (0, 0, 0).

As triangulações permitem a representação de superfícies e objetos por meio de pedaços triangulares que não se intersectam. A melhor triangulação é feita considerando-se as menores distâncias possíveis entre os pontos de amostra da superfície representada graficamente.

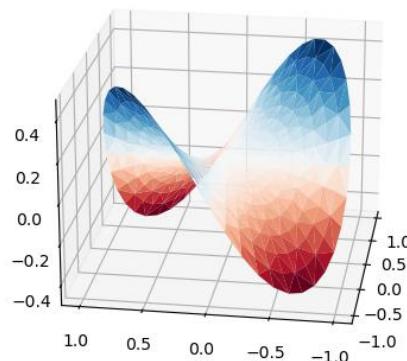
Utilizando-se a função `plot_trisurf`, podemos representar a triangulação de uma superfície quando conhecemos as expressões que definem as variáveis x, y e z.

```
import matplotlib.pyplot as plt
import numpy as np

n_raio = 10
n_angulos = 48
raio = np.linspace(0.125, 1.0, n_raio)
angulo = np.linspace(0, 2*np.pi, n_angulos, endpoint = False)[..., np.newaxis]

x = np.append(0, (raio*np.cos(angulo)).flatten())
y = np.append(0, (raio*np.sin(angulo)).flatten())
z = np.sin(-x*y)

ax = plt.figure().add_subplot(projection = '3d')
ax.plot_trisurf(x, y, z, linewidth = 0.2, cmap = 'RdBu')
plt.show()
```



Utilizando as coordenadas de alguns pontos da superfície de um terreno (latitude e longitude ou coordenadas relativas UTM) podemos representar suas triangulações por meio da função `plot_trisurf`.

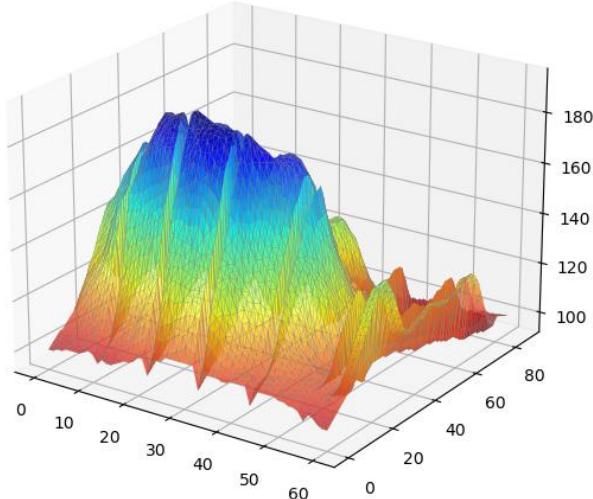
```
import numpy as np
import matplotlib.pyplot as plt

vertices = np.array(np.loadtxt('C:/dados/volcano.txt', int))
x = vertices[:,0]
y = vertices[:,1]
z = vertices[:,2]

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

ax.plot_trisurf(x, y, z, cmap = 'jet_r', edgecolor = 'grey', linewidth = 0.15, alpha = 0.7)

plt.show()
```



Dados disponíveis em: <https://python-graph-gallery.com/371-surface-plot>

As representações gráficas das superfícies de alguns objetos e de partes do corpo humano podem ser feitas por meio de triangulações da mesma forma mostrada de um terreno. Podemos utilizar os arquivos que possuem as coordenadas dos vértices e as referências de faces em formatos como OBJ ou PLY (Polygon File Format).

Vamos ver o código Python para representar um objeto com as informações em formato PLY. Veja o exemplo mostrado de um cubo em formato PLY.

```

ply
format ascii 1.0
comment created by platopoly
element vertex 8
property float32 x
property float32 y
property float32 z
element face 6
property list uint8 int32 vertex_indices
end_header
-1 -1 -1 → Coordenadas do vértice 0.
1 -1 -1
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
4 0 1 2 3 → Primeira face: 4 vértices de índices 0, 1, 2 e 3.
4 5 4 7 6
4 6 2 1 5
4 3 7 4 0
4 7 3 2 6
4 5 1 0 4

```

No cabeçalho temos as informações dos números de vértices e de faces dos objetos. Logo após o fim do cabeçalho aparecem as sequências das coordenadas dos vértices e as codificações das faces. Por exemplo, a primeira face tem 4 vértices com índices 0, 1, 2 e 3. O formato de arquivo OBJ tem informações similares.

A biblioteca plyfile tem as funções para leitura de arquivos PLY. As primeiras informações usadas são os números de vértices e de faces do objeto.

```

from plyfile import PlyData
import numpy as np
import matplotlib.pyplot as plt

plydata = PlyData.read('C:/dados/galleon.ply')

```

```

with open('C:/dados/galleon.ply', 'rb') as f:
    plydata = PlyData.read(f)

plydata.elements[0].name
plydata.elements[0].data[0]
nr_vertices = plydata.elements[0].count
nr_faces = plydata.elements[1].count

```

Os próximos comandos fazem as atribuições dos vértices, das faces e a representação da triangulação por meio da função `plot_trisurf`.

```

vertices = np.array([plydata['vertex'][k] for k in range(nr_vertices)])

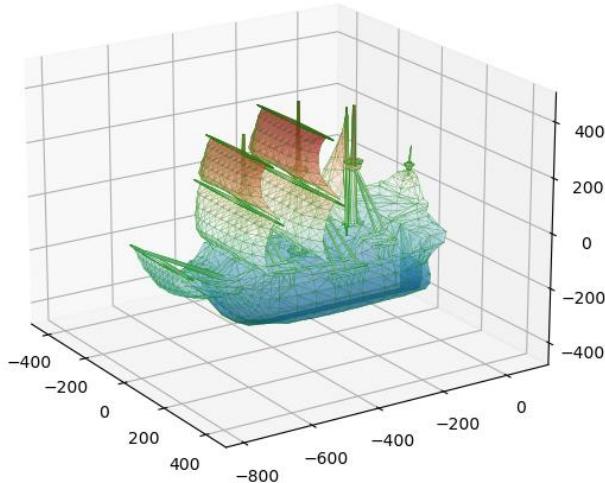
x, y, z = zip(*vertices)

faces = [plydata['face'][k][0] for k in range(nr_faces)]
ax = plt.figure().add_subplot(projection = '3d')

ax.plot_trisurf(x, y, z, triangles = faces, cmap = 'RdBu_r', edgecolor = 'green',
                 linewidth = 0.1, alpha = 0.5)

plt.show()

```



Dados disponíveis em: <https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html>

Usando as funções para manter as proporções entre as medidas dos eixos, temos a representação gráfica ideal de cada arquivo PLY:

```

def set_axes_equal(ax: plt.Axes):
    limits = np.array([ax.get_xlim3d(), ax.get ylim3d(), ax.get_zlim3d()])
    origin = np.mean(limits, axis=1)
    radius = 0.5 * np.max(np.abs(limits[:, 1] - limits[:, 0]))
    _set_axes_radius(ax, origin, radius)

def _set_axes_radius(ax, origin, radius):
    x, y, z = origin
    ax.set_xlim3d([x - radius, x + radius])
    ax.set_ylim3d([y - radius, y + radius])
    ax.set_zlim3d([z - radius, z + radius])

set_axes_equal(ax)

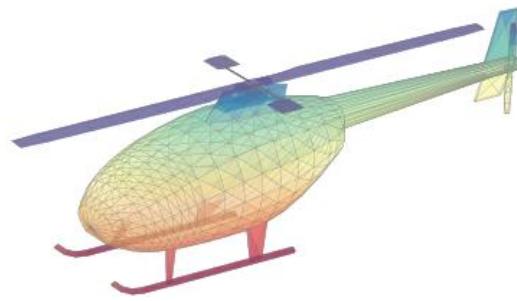
```

Para mostrar apenas o objeto, sem os eixos principais e auxiliares, basta usar o atributo `off` na propriedade `axis` da figura. Veja o exemplo dos dados do helicóptero sem os eixos:

```

plydata = PlyData.read('C:/dados/chopper.ply')
plt.axis('off')

```



As imagens de alguns objetos representados com as bibliotecas mostradas em Python, com a triangulação das coordenadas de vértices dos objetos são os seguintes: Mug, tennis_shoe, teapot, hind, footbones, airplane e big_porsche.



Dados disponíveis em: <https://people.sc.fsu.edu/~jburkardt/data/ply/pl.html>

Podemos utilizar dois arquivos com as informações dos objetos para representá-los graficamente. Nestes casos, precisamos indicar que as faces têm as coordenadas dos vértices com índices indicados no arquivo de faces, ou seja: `vertices[faces]`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

vertices = np.loadtxt('C:/dados/vertices_hind.txt')
faces = np.loadtxt('C:/dados/faces_hind.txt', int)

facesc = np.array(vertices[faces])

fig = plt.figure()
ax = fig.add_subplot(projection = '3d')

ax.add_collection3d(Poly3DCollection(facesc, facecolors = 'green', edgecolors = 'grey',
alpha = 0.25, linewidth = 0.1))

ax.set_xlim3d(np.min(vertices[:,0]), np.max(vertices[:,0]))
```

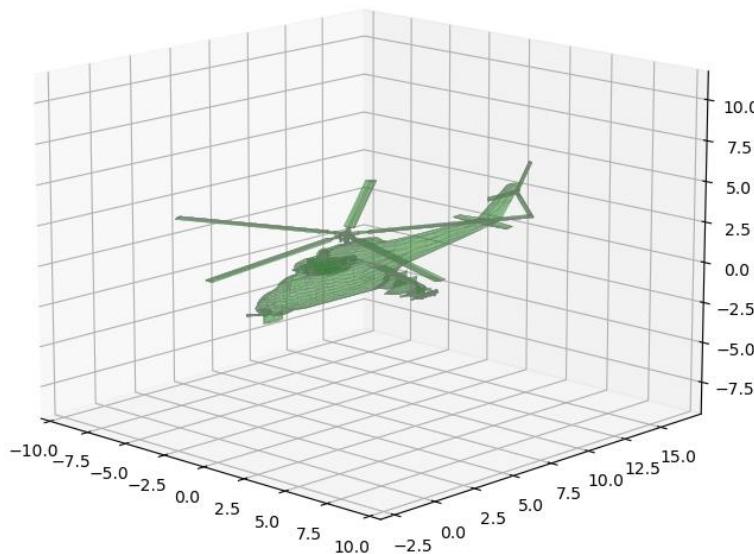
```

ax.set_ylim3d(np.min(vertices[:,1]), np.max(vertices[:,1]))
ax.set_zlim3d(np.min(vertices[:,2]), np.max(vertices[:,2]))

plt.show()

```

Usando as funções para manter as proporções entre as medidas dos eixos, temos a representação gráfica do objeto hind.



Atividade 5

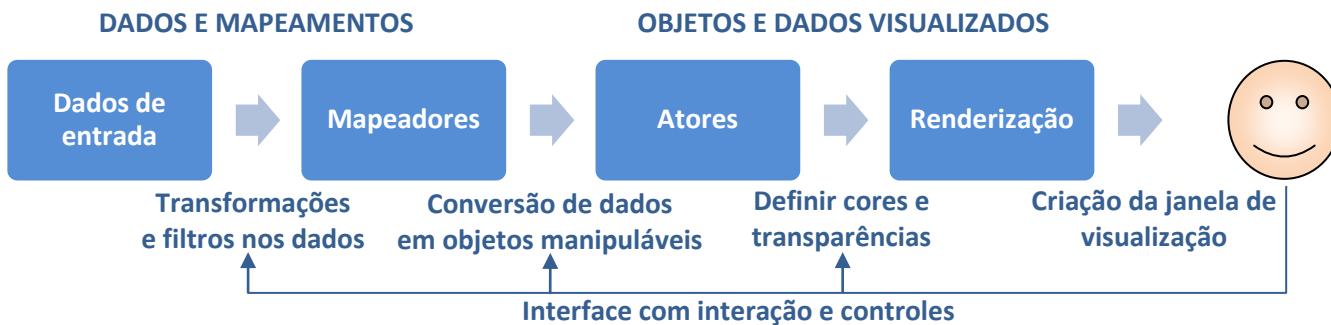
- 5.1. Utilize a função `add_collection3d` para representar graficamente uma pirâmide reta de base quadrada, com altura 7 e coordenadas dos dois vértices consecutivos da base: A(1, 1, 1) e B(2, 5, 1).
- 5.2. Utilize a função `add_collection3d` para representar graficamente uma pirâmide oblíqua de base quadrada com vértices consecutivos da base A(2, 1, 2) e B(4, 5, 2) e vértice principal E(10, 4, 6).
- 5.3. Utilize a função `add_collection3d` para representar graficamente um octaedro regular com aresta da seção equatorial com coordenadas A(1, 5, 4) e B(3, 1, 4).
- 5.4. Escolha um conjunto de coordenadas de uma superfície para representá-la com a função `trisurf`.
- 5.5. Escolha três objetos 3D de arquivos com formato PLY para representá-los com as funções de triangulações da biblioteca Poly3DCollection.

6. Modelos de Iluminação

Uma parte importante da visualização científica envolve a escolha da iluminação dos objetos no ambiente virtual que aparece para os observadores. Vamos estudar os tipos de iluminação dos objetos utilizando o sistema **VTK** (Visualization Toolkit). Este sistema tem código aberto orientado a objetos para computação gráfica 3D, visualização e processamento de imagens. Este sistema está programado em C++, mas tem suporte para funcionar em Python e Java. A utilização da programação em VTK permite a criação de aplicativos complexos, prototipagem rápida e scripts simples.

O sistema VTK fornece uma variedade de representações de dados, incluindo conjuntos de pontos não organizados, dados poligonais, imagens, poliedros e outros sólidos. Muitos filtros de processamento de dados estão disponíveis para operar no VTK, desde a convolução da imagem até a Triangulação de Delaunay. O modelo de renderização do VTK suporta também poligonais e funções 2D.

O pipeline da construção de uma janela de visualização com o sistema VTK está apresentado a seguir.



O código mostrado a seguir ilustra todos os estágios do pipeline do VTK para mostrar a representação de um cilindro circular reto utilizando as bibliotecas vtkmodules de interação, cores, cilindro, eixos e renderização.

```

from vtkmodules.vtkCommonColor import vtkNamedColors
from vtkmodules.vtkFiltersSources import vtkCylinderSource
from vtkmodules.vtkRenderingAnnotation import vtkAxesActor
from vtkmodules.vtkRenderingCore import (
    vtkActor,
    vtkPolyDataMapper,
    vtkRenderWindow,
    vtkRenderWindowInteractor,
    vtkRenderer
)

def main():
    colors = vtkNamedColors()
    colors.SetColor("BkgColor", [0.95, 0.95, 1, 0])           → Dados de entrada: cor do fundo e cilindro
    cylinder = vtkCylinderSource()                         → Mapeamento da geometria do cilindro
    cylinder.SetResolution(30)

    cylinderMapper = vtkPolyDataMapper()                   → Criação do ator
    cylinderMapper.SetInputConnection(cylinder.GetOutputPort())

    cylinderActor = vtkActor()                           → Adiciona as propriedades primitivas do cilindro
    cylinderActor.SetMapper(cylinderMapper)
    cylinderActor.GetProperty().SetColor(colors.GetColor3d("Yellow"))
    cylinder.SetRadius(0.5)
    cylinder.SetHeight(1.5)
    cylinderActor.SetPosition(2, -1, 1.5)                → Propriedades do cilindro (ator)
    cylinderActor.RotateZ(-30.0)
    cylinderActor.RotateX(-30.0)

    ren = vtkRenderer()
    renWin = vtkRenderWindow()
    renWin.AddRenderer(ren)                             → Renderização (janela e interação com o usuário)
    iren = vtkRenderWindowInteractor()
    iren.SetRenderWindow(renWin)

    ren.AddActor(cylinderActor)
    axes = vtkAxesActor()                            → Adição do cilindro e de mais um ator: eixos
    ren.AddActor(axes)
    ren.SetBackground(colors.GetColor3d("BkgColor"))
    renWin.SetSize(500, 500)

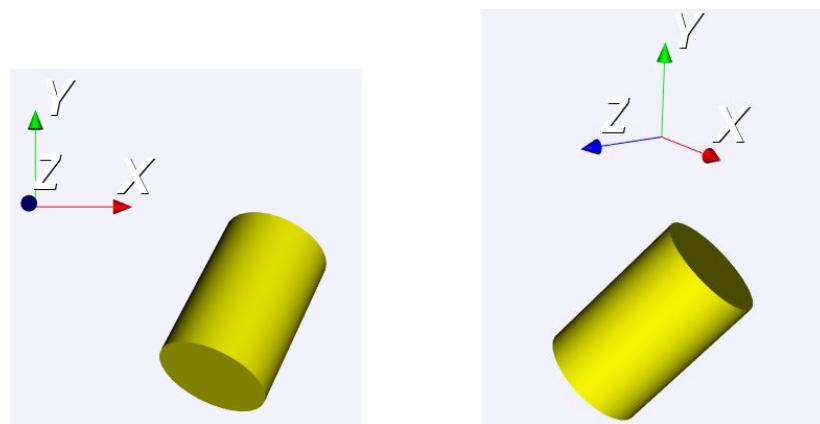
    iren.Initialize()
    ren.ResetCamera()
    ren.GetActiveCamera().Zoom(1.2)                    → Iniciar a câmera: zoom e renderização da cena
    renWin.Render()
    iren.Start()

if __name__ == '__main__':
    main()

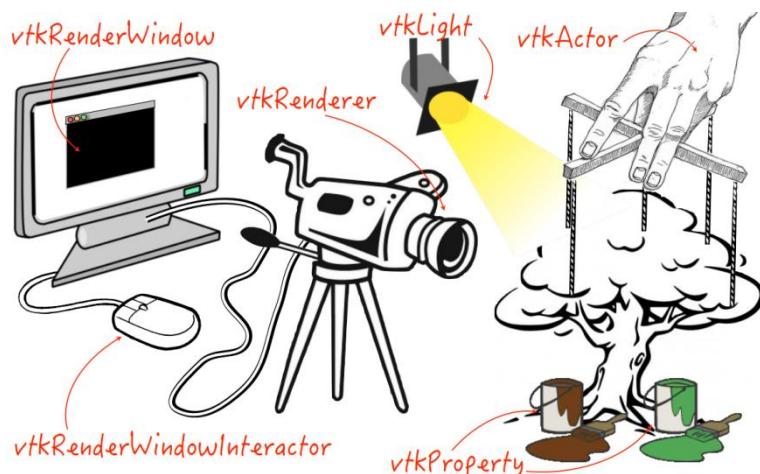
```

O código Python demonstra o pipeline VTK para renderizar um cilindro. Ele importa classes de vtkmodules e configura a cor de fundo para amarelo. Cria um fonte de cilindro com resolução de 30 segmentos. Define um mapeador para a geometria do cilindro e um ator para ele. Define propriedades para o ator, incluindo cor amarela, raio de 0.5 e altura de 1.5. Define a posição do cilindro no espaço 3D. Adiciona o cilindro e um ator de eixos ao renderer. Configura o renderer para usar a cor de fundo definida e define o tamanho da janela de renderização. Inicializa o interator, configura a câmera para zoom e renderiza a cena. Por fim, verifica se o script é executado diretamente e chama a função main().

O resultado aparece na janela de visualização e interação. As coordenadas X e Y funcionam como as coordenadas da tela do dispositivo, e a coordenada Z “sai da tela”. Esta mesma configuração usada na Computação Gráfica será usada nos ambientes que vamos programar em Realidade Aumentada e Realidade Virtual.



Resumindo, temos o seguinte esquema com todos os elementos para renderização de objetos em uma cena com o sistema VTK:



Fonte: <https://www.cs.purdue.edu/homes/cs530/>

O sombreamento revela as formas de objetos 3D por meio de sua interação com a luz. O sombreamento cria cores em função dos seguintes elementos: propriedade das superfícies, direções das retas normais das superfícies e a posição da luz. O sombreamento faz uma espécie de controle da visualização de cada superfície representada.

O processo de iluminação utilizado na Computação Gráfica foi desenvolvido pelo pesquisador Bui Tuong Phong. Trata-se de uma combinação de três tipos de luzes: ambiente L_a , difusa L_d e specular L_s .

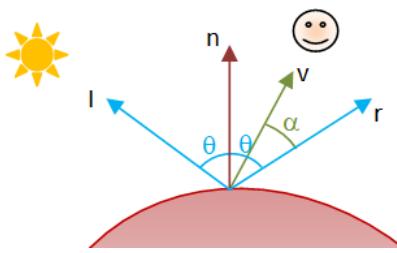
Vamos começar mostrando os elementos do tipo de luz ambiente. A quantidade de luz ambiente refletida pode ser determinada por meio das quantidades de cada cor básica do objeto (R, G, B):

$$I_a = \begin{pmatrix} I_{aR} \\ I_{aG} \\ I_{aB} \end{pmatrix}.$$

Devemos considerar também as propriedades do material do objeto para representar sua respectiva iluminação, ou seja:

$$k_a = \begin{pmatrix} k_{aR} \\ k_{aG} \\ k_{aB} \end{pmatrix}.$$

Outro elemento que deve ser considerado é da intensidade de emissão de luz, que aparece em objetos que emitem luz. A composição de iluminação de uma superfície leva em conta os 4 vetores indicados na figura a seguir.



Vetor normal n: perpendicular à superfície e direcionado para fora da superfície.

Vetor de visualização v: aponta para a direção do visualizador.

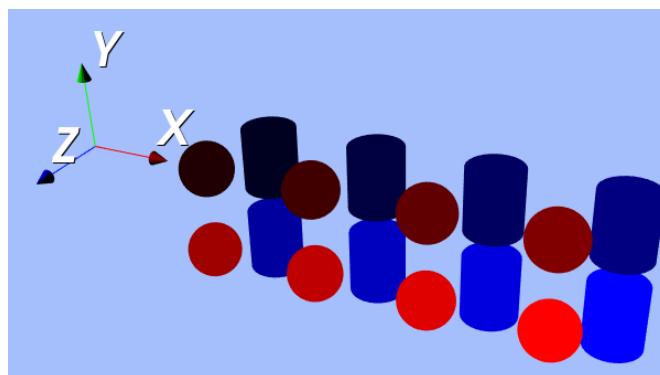
Vetor de luz I: aponta para a fonte de luz.

Vetor de reflexão r: indica a direção da reflexão pura da luz do vetor.

A luz ambiente gera iluminação constante da superfície, e depende apenas do material e da cor do objeto:

$$L_a = k_a \cdot I_a.$$

Usando o sistema VTK, vamos visualizar 8 intensidades de luz ambiente para a representação de um cilindro ao lado de uma esfera.



```
from vtkmodules.vtkCommonColor import vtkNamedColors
from vtkmodules.vtkFiltersSources import vtkSphereSource
from vtkmodules.vtkFiltersSources import vtkCylinderSource
from vtkmodules.vtkRenderingAnnotation import vtkAxesActor
from vtkmodules.vtkRenderingCore import (
    vtkActor,
    vtkLight,
    vtkPolyDataMapper,
    vtkRenderWindow,
    vtkRenderWindowInteractor,
    vtkRenderer
)
def main():
    colors = vtkNamedColors()
    colors.SetColor('bkg', [0.65, 0.75, 0.99, 0])
    sphere = vtkSphereSource()
    sphere.SetThetaResolution(100)
    sphere.SetPhiResolution(50)
    sphere.SetRadius(0.3)
    cylinder = vtkCylinderSource()
    cylinder.SetResolution(30)
    cylinder.SetRadius(0.3)
    cylinder.SetHeight(0.7)
    —————> Dados de entrada: cor do fundo, cilindros e esferas
```

```

sphereMapper = vtkPolyDataMapper()
cylinderMapper = vtkPolyDataMapper()
sphereMapper.SetInputConnection(sphere.GetOutputPort())
cylinderMapper.SetInputConnection(cylinder.GetOutputPort())

quantidade = 8
spheres = list()
cylinders = list()
ambient = 0.125 → Neste exemplo, vamos visualizar apenas a luz ambiente
diffuse = 0.0
specular = 0.0
position = [1.5, 0, 0]
position1 = [2, 0, -0.5]
for i in range(0, quantidade):
    spheres.append(vtkActor())
    spheres[i].SetMapper(sphereMapper)
    spheres[i].GetProperty().SetColor(colors.GetColor3d('Red'))
    spheres[i].GetProperty().SetAmbient(ambient)
    spheres[i].GetProperty().SetDiffuse(diffuse)
    spheres[i].GetProperty().SetSpecular(specular)
    spheres[i].AddPosition(position) → Loop para criar os Atores e definir iluminação de cada par
    cylinders.append(vtkActor())
    cylinders[i].SetMapper(cylinderMapper)
    cylinders[i].GetProperty().SetColor(colors.GetColor3d('Blue'))
    cylinders[i].GetProperty().SetAmbient(ambient)
    cylinders[i].GetProperty().SetDiffuse(diffuse)
    cylinders[i].GetProperty().SetSpecular(specular)
    cylinders[i].AddPosition(position1)
    ambient += 0.125
    position[0] += 1.25
    position1[0] += 1.25
    if i == 3:
        position[0] = 1.5
        position[1] = -1
        position1[0] = 2
        position1[1] = -1

ren = vtkRenderer()
axes = vtkAxesActor()
ren.AddActor(axes)
renWin = vtkRenderWindow() → Renderização
renWin.AddRenderer(ren)
iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

for i in range(0, quantidade):
    ren.AddActor(spheres[i])
    ren.AddActor(cylinders[i])

ren.SetBackground(colors.GetColor3d('bkg'))
renWin.SetSize(940, 480)
renWin.SetWindowName('AmbientSpheres')

light = vtkLight()
light.SetFocalPoint(1.8, 0.6, 0)
light.SetPosition(0.8, 1.6, 1) → Posição da fonte de luz
ren.AddLight(light)

iren.Initialize()
renWin.Render()
iren.Start()
if __name__ == '__main__':
    main()

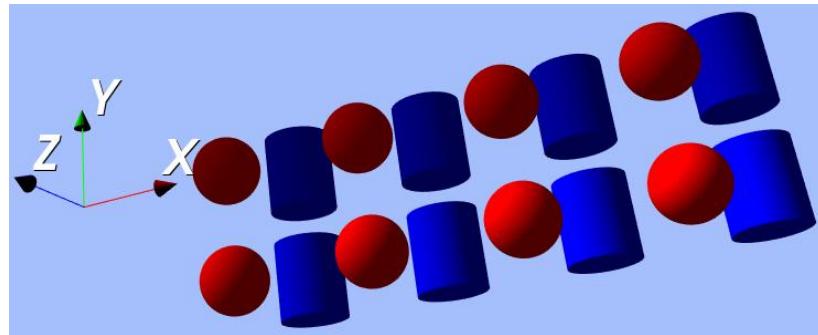
```

A luz difusa surge da suposição de que a luz de qualquer direção é refletida uniformemente em todas as direções. Este tipo de iluminação depende da cor do objeto, da posição da luz e da orientação da face onde está o ponto (vetor normal). Este tipo de luz cria o efeito degradé nas superfícies.

Considerando que os vetores \vec{n} e \vec{l} são normalizados, temos a luz difusa definida como:

$$L_d = k_d \cdot \cos(\theta) \cdot I_d = k_d \cdot (\vec{n} \cdot \vec{l}) \cdot I_d.$$

Quando $\vec{n} \cdot \vec{l} < 0$, o ponto está na parte escura do objeto. Utilize o mesmo arquivo do código VTK usado na visualização da luz ambiente para gerar as 8 fases de variação da luz difusa.

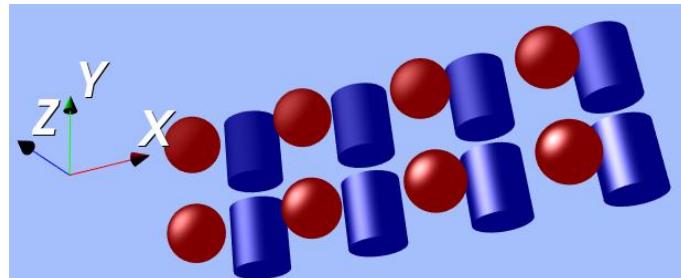


A luz specular é a componente que produz o ponto de brilho mais acentuado da superfície. Os componentes desta luz dependem da cor, da posição da fonte de luz, da posição do observador, da posição de cada ponto e da orientação da superfície (vetor normal). O brilho intenso gerado por esta luz tem a cor da luz e não a cor do objeto.

O parâmetro s define a intensidade do brilho da luz specular. Considerando que os vetores \vec{r} e \vec{v} são normalizados, temos a luz specular de intensidade s definida como:

$$L_s = k_s \cdot \cos^s(\alpha) \cdot I_s = k_s \cdot (\vec{r} \cdot \vec{v})^s \cdot I_s.$$

Utilize o mesmo arquivo do código VTK usado na visualização da luz ambiente e na luz difusa para gerar as 8 fases de variação da luz specular. Neste caso, temos mais dois elementos para definir: a cor e a intensidade da luz:

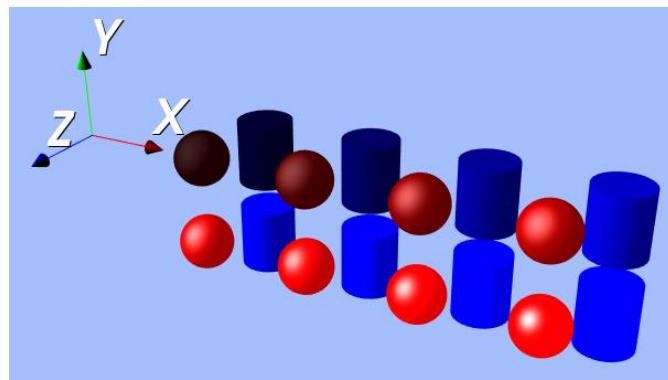


```
#Valores iniciais:
ambient = 0.8
specular = 0.125
specularPower = 1

#Valores para o loop de esferas e cilindros:
spheres[i].GetProperty().SetSpecular(specular)
spheres[i].GetProperty().SetSpecularPower(specularPower)
spheres[i].GetProperty().SetSpecularColor(colors.GetColor3d('White'))
cylinders[i].GetProperty().SetSpecular(specular)
cylinders[i].GetProperty().SetSpecularPower(specularPower)
cylinders[i].GetProperty().SetSpecularColor(colors.GetColor3d('White'))

#Valores dos incrementos dos parâmetros:
specular += 0.125
specularPower += 0.5
```

Agora podemos combinar estes tipo de luz para verificar os melhores parâmetros para as representações da esfera e do cilindro. Faça o loop com o código usado anteriormente variando os incrementos dos parâmetros dos três tipos de luzes.

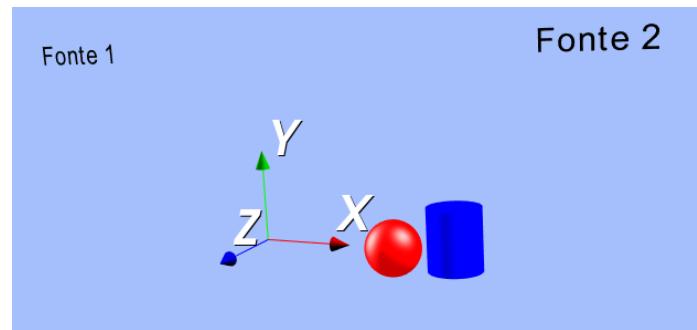


Podemos definir mais de uma fonte de luz no ambiente VTK. Inserindo as propriedades de iluminação nos atores `cylinderActor` e `sphereActor`, definimos mais um ator: `atext` (para mostrar a origem de cada fonte). O código mostrado a seguir é de uma das fontes de luz que está no ponto (-3, 2, 0). A outra fonte de luz deste projeto está no ponto (3, 2, 0). O código do texto da Fonte2 é similar ao mostrado da Fonte1.

```
atext = vtkVectorText()
atext.SetText('Fonte 1')
textMapper = vtkPolyDataMapper()
textMapper.SetInputConnection(atext.GetOutputPort())
textActor = vtkFollower()
textActor.SetMapper(textMapper)
textActor.SetScale(0.2, 0.2, 0.2)
textActor.AddPosition(-3, 2, 0)
```

Na etapa de renderização, inserimos todos os atores da cena:

```
ren.AddActor(axes)
ren.AddActor(cylinderActor)
ren.AddActor(sphereActor)
ren.AddActor(textActor)
ren.AddActor(textActor1)
```



Depois adicionamos as duas fontes de iluminação da cena:

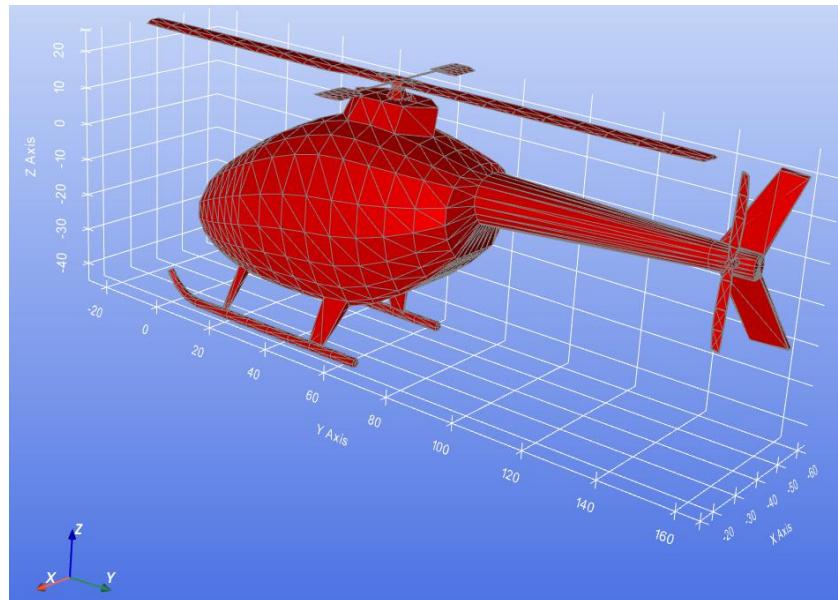
```
light = vtkLight()
light.SetFocalPoint(1.8, 0.6, 0)
light.SetPosition(0, 2, 5)
ren.AddLight(light)

light1 = vtkLight()
light1.SetFocalPoint(1.8, 0.6, 0)
light1.SetPosition(0, 2, -3)
ren.AddLight(light1)
```

O **PyVista** é um módulo auxiliar para o Visualization Toolkit (VTK) que adota uma abordagem simplificada da interface com o VTK por meio da biblioteca NumPy e acesso direto aos arrays. Esta biblioteca fornece uma interface bem documentada de funções e atributos VTK e Python, resultando em visualizações de objetos do VTK para facilitar a prototipagem rápida, análise e integração visual de conjuntos de dados referenciados espacialmente.

Este módulo pode ser usado para plotagem científica de apresentações e trabalhos de pesquisa, bem como um módulo de suporte para outros módulos Python dependentes de malhas geométricas 2D e 3D. No exemplo mostrado a seguir, vamos utilizar o arquivo `chopper.ply` para mostrar o código simplificado de VTK dentro da biblioteca PyVista.

Utilizamos o comando `add_mesh` (adicionar malha) para inserir o objeto lido do arquivo `ply`. Os atributos de iluminação estão todos neste comando de malha. Além disso, as duas fontes de luz foram inseridas com o comando `add_light`.



```
import pyvista
import pyvista as pv

filename = 'C:/dados/chopper.ply'
reader = pyvista.get_reader(filename)
mesh = reader.read()

p = pv.Plotter(lichting = 'none', window_size = [1000, 1000])
p.show_grid()
p.show_axes()

light = pv.Light(position = (-10, 1, 1), light_type = 'scene light')
p.add_light(light)
light = pv.Light(position = (10, 1, 1), light_type = 'scene light')
p.add_light(light)

p.set_background('royalblue', top = 'aliceblue')
p.add_mesh(mesh, color = 'Red', show_edges = True, edge_color = 'grey', ambient = 0.3,
           diffuse = 0.5, specular = 0.5, specular_power = 15)

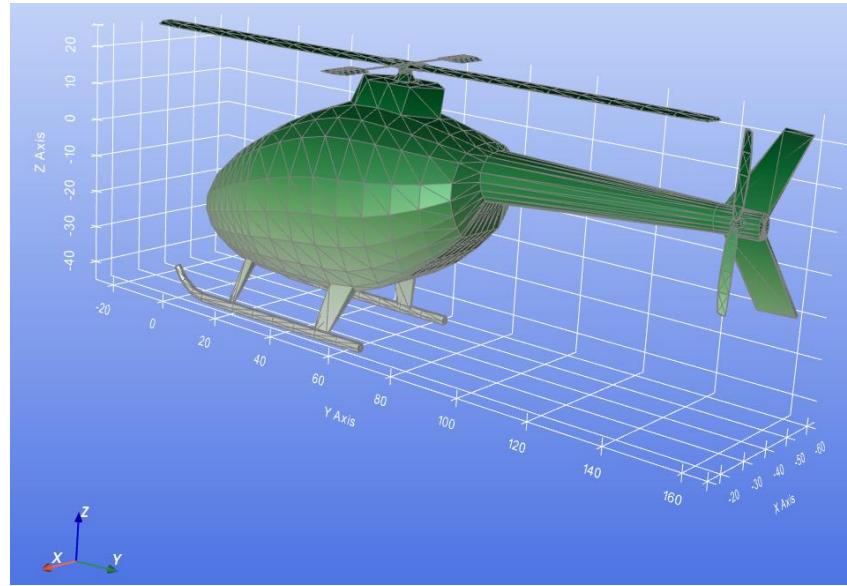
p.show()
```

As instâncias de iluminação do `pyvista` são de três tipos: faróis (head light), luzes da câmera (camera light) e luzes da cena (scene light). Os faróis sempre brilham ao longo do eixo da câmera, as luzes da câmera têm uma posição fixa em relação à câmera e as luzes da cena são posicionadas em relação à cena, de modo que o movimento ao redor da câmera não interfere na iluminação da cena. Modifique os atributos `light_type` para visualizar o objeto com estes outros tipos de iluminação.

As luzes têm uma posição e um ponto focal que definem o eixo da luz. A cor da luz pode ser definida de acordo com os componentes ambiente, difuso e especular. O brilho pode ser definido com a propriedade de intensidade (intensity) que vamos definir adiante.

Podemos inserir mapas de cores nos objetos, como fizemos anteriormente com a biblioteca matplotlib. Neste caso, precisamos definir qual variável será usada para controlar a variação de cores:

```
p.add_mesh(mesh, cmap = 'Greens', scalars = mesh.points[:, 2], show_scalar_bar = False,
           show_edges = True, edge_color = 'grey', ambient = 0.3, diffuse = 0.5, specular = 0.5,
           specular_power = 15)
```



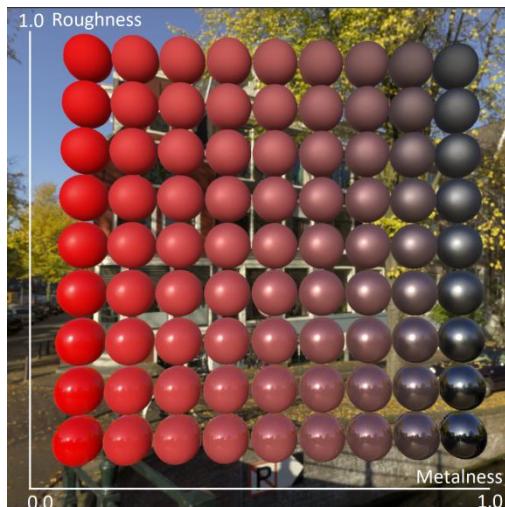
O atributo de transparência opacity pode ser usado no comando `add_mesh`. Os materiais **PBR** (Physically Based Rendering) são usados na Computação Gráfica para representar as malhas que devem receber iluminação realista. O uso de materiais PBR indica que os objetos representados possuem propriedades visuais com resultados muito próximos do mundo real. Estes objetos com propriedades PBR são usados na programação de jogos e ambientes virtuais modernos pois são considerados como as melhores aproximações para cenários do mundo real.

Utilizamos os atributos PBR com base na física por meio das propriedades: cor, metalicidade (metallic) e rugosidade (roughness).

Na física, a **metalicidade** diz se uma superfície tem material condutor de eletricidade ou dielétrico (que dificulta a passagem da corrente elétrica). Nos materiais representados em PBR, essa propriedade afeta o quanto a superfície reflete o ambiente ao redor. Quando a metalicidade é 0, a cor do objeto fica totalmente visível e o material parece com plástico ou cerâmica. Quando a metalicidade vale 0.5, o objeto parece com metal pintado. Quando a metalicidade vale 1, a superfície perde quase completamente a cor e reflete apenas o que existir em seus arredores. Quando a metalicidade vale 1 e a rugosidade vale 0, a superfície fica parecida com um espelho.

A rugosidade representa o nível de aspereza ou suavidade de uma superfície. As superfícies ásperas dispersam a luz em mais direções que as suaves, tornando os reflexos de luz desfocados. Quando a rugosidade de um objeto vale 0, os reflexos de luz são nítidos. Quando a rugosidade vale 0.5, os reflexos de luz ficam desfocados.

Na imagem mostrada a seguir, temos as combinações dos valores destes atributos PBR de metalicidade e rugosidade em esferas vermelhas.



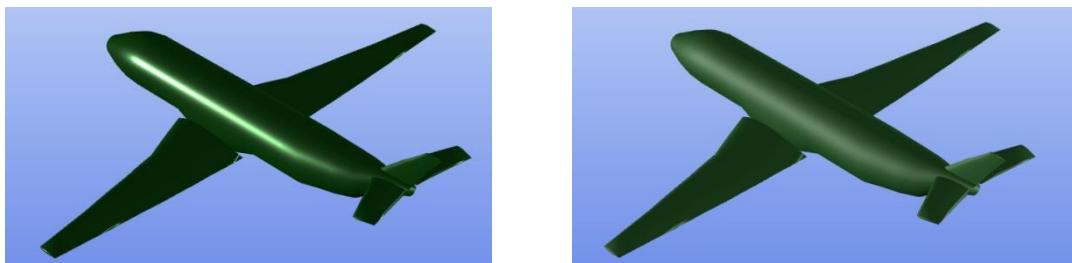
Disponível em: <https://docs.microsoft.com/pt-br/azure/remote-rendering/overview/features/pbr-materials>

Com os dados do arquivo airplane.ply, podemos atribuir os valores dos atributos `metallic` e `roughness` para definir a melhor representação gráfica deste objeto com a iluminação definida. No primeiro caso, temos a metalicidade maior do que a rugosidade, resultando no aspecto de um objeto condutor.

```
p.add_mesh(mesh, color = 'Green', show_edges = False, ambient = 0.3, diffuse = 0.5,
          specular = 0.5, specular_power = 15, opacity = 1, metallic = 0.6, roughness = 0.3,
          pbr = True)
```

No segundo caso, com a rugosidade maior do que a metalicidade, o objeto fica com aspecto de plástico.

```
p.add_mesh(mesh, color = 'Green', show_edges = False, ambient = 0.3, diffuse = 0.5,
          specular = 0.5, specular_power = 15, opacity = 1, metallic = 0.3, roughness = 0.6,
          pbr = True)
```



As fontes de iluminação definidas no VTK podem ser direcionais (ou seja, uma fonte pontual colocada em uma posição infinitamente distante) ou posicionais. As luzes posicionais têm propriedades adicionais que descrevem a geometria e a distribuição espacial da luz. As propriedades `cone_angle` e `exponent` definem a forma do feixe de luz e a distribuição angular da intensidade da luz dentro desse feixe.

As luzes posicionais com `cone_angle` menor que 90° são conhecidas como holofotes. Os holofotes são unidirecionais e utilizam as propriedades de modelagem do feixe, ou seja, expoente e atenuação. As luzes posicionais sem holofotes, no entanto, agem como fontes pontuais localizadas na posição real da luz, brilhando em todas as direções do espaço. Eles exibem atenuação com a distância da fonte, mas seu feixe é isotrópico no espaço. Em contraste, as luzes direcionais atuam como fontes pontuais infinitamente distantes, portanto, são unidirecionais, mas não atenuam.

Com as luzes direcionadas, podemos criar cenários de iluminação complexos. Por exemplo, você pode posicionar uma luz diretamente acima de um ator, para criar uma sombra diretamente abaixo dele. No exemplo mostrado a seguir, calculamos os pontos extremos do conjunto de coordenadas x, y e z do objeto para construir o plano e definir a posição e o ponto focal da fonte de luz. A propriedade `enable_shadows()` permite a criação das sombras no ambiente.

```
import pyvista
```

```

import pyvista as pv
import numpy as np

filename = 'C:/dados/chopper.ply'
reader = pyvista.get_reader(filename)
mesh = reader.read()

p = pv.Plotter(lighting = None, window_size = [1000, 1000])
p.show_axes()

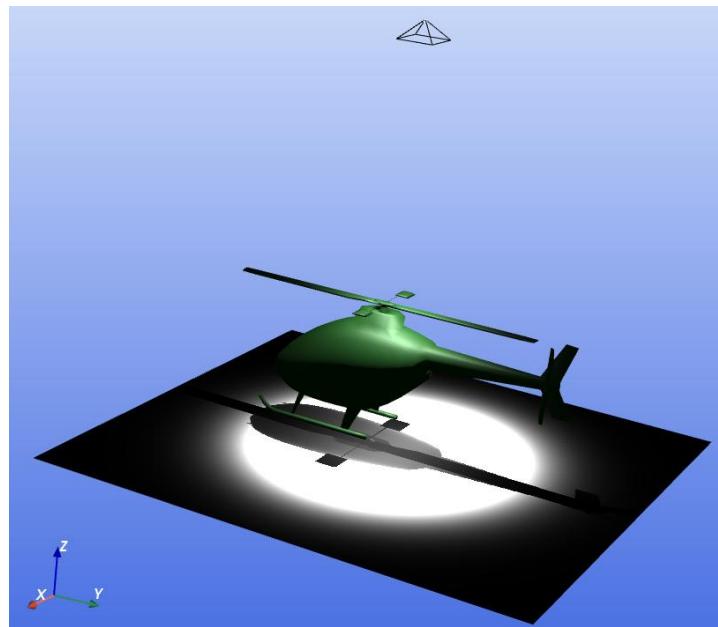
maxx= np.max(mesh.points[:, 0])
maxy= np.max(mesh.points[:, 1])
minx= np.min(mesh.points[:, 0])
miny= np.min(mesh.points[:, 1])
minz= np.min(mesh.points[:, 2])
maxz= np.max(mesh.points[:, 2])

light = pv.Light(position = [(maxx + minx)/2, (maxy + miny)/2, maxz + 150],
                 focal_point = [(maxx + minx)/2, (maxy + miny)/2, 0], show_actor = True,
                 positional = True, cone_angle = 45, exponent = 50, intensity = 30)
p.add_light(light)

p.set_background('royalblue', top = 'aliceblue')
p.add_mesh(mesh, color = 'Green', show_edges = False, ambient = 0.3, diffuse = 0.5,
           specular = 1, specular_power = 15, opacity = 1, metallic = 0.3, roughness = 0.6,
           pbr = True)

grid = pv.Plane(i_size = 5*(maxx - minx), j_size = 2*(maxy + miny),
                center = [(maxx + minx)/2, (maxy + miny)/2, minz - 10])
p.add_mesh(grid, color = 'white')
p.enable_shadows()
p.show()

```

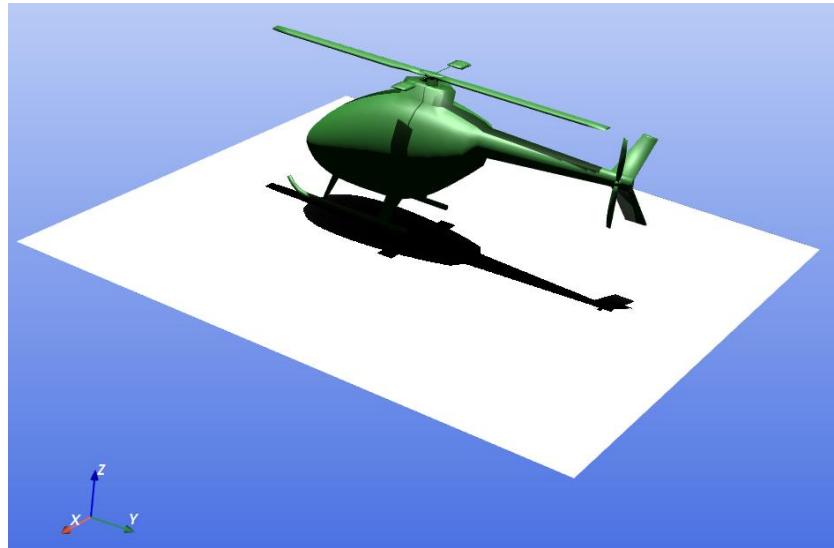


Para usarmos a luz direcional, basta definir o atributo `positional = None` sem valores atribuídos para `cone_angle` e `exponent`.

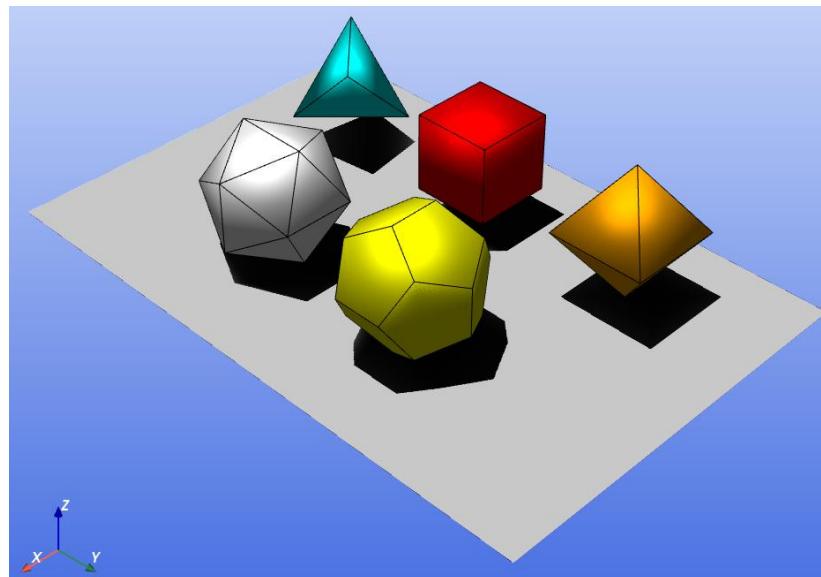
```

light = pv.Light(position = [(maxx + minx)/2, (maxy + miny)/2, maxz + 150],
                  focal_point = [(maxx + minx)/2, (maxy + miny)/2, 0], positional = None, intensity = 30)

```



Crie as iluminações para outros objetos com os comandos mostrados. Usando os comandos para representar os sólidos de Platão, podemos criar mais uma cena com iluminação direcional e com sombreamento suave (smooth_shading).



```

import pyvista as pv

kinds = ['tetrahedron', 'cube', 'octahedron', 'dodecahedron', 'icosahedron']
centers = [(-1, 0, 0), (-1, 1, 0), (-1, 2, 0), (0, 1.5, 0), (0, 0.5, 0)]

solids = [pv.PlatonicSolid(kind, radius = 0.4, center = center) for kind, center in zip(kinds, centers)]
colors = ['aqua', 'red', 'orange', 'yellow', 'white']

p = pv.Plotter(lichting = 'none', window_size = [1000, 1000])
p.set_background('royalblue', top = 'aliceblue')

for ind, solid in enumerate(solids):
    p.add_mesh(solid, colors[ind], ambient = 0.3, smooth_shading = True, show_edges = True,
               diffuse = 0.8, specular = 0.5, specular_power = 2)

p.add_floor('-z', lighting = True, color = 'white', pad = 0.4)
p.show_axes()

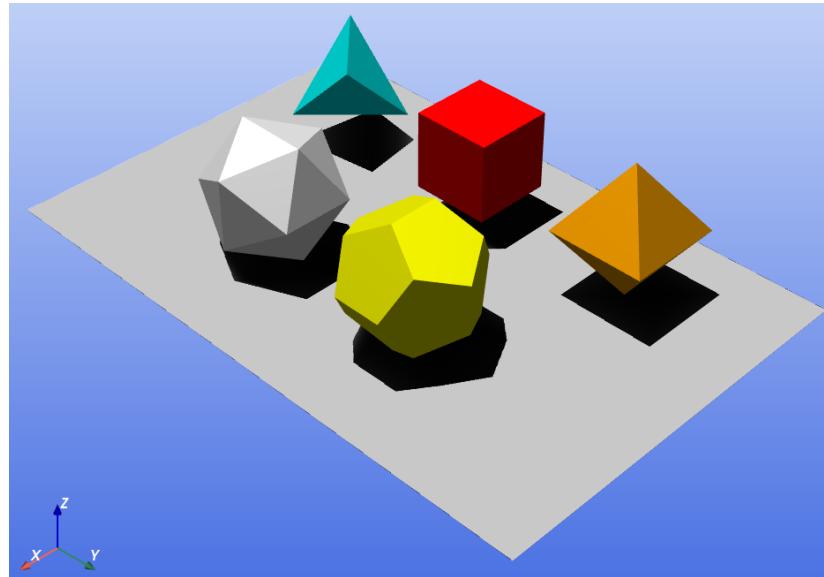
p.add_light(pv.Light(position = (1, -1, 5), focal_point = (0, 0, 0), color = 'white',
                     intensity = 0.8))

```

```
p.enable_shadows()
p.show()
```

A cena com os sólidos sem o sombreamento suave e sem as arestas pode ser feita com os comandos:

```
for ind, solid in enumerate(solids):
    p.add_mesh(solid, colors[ind], ambient = 0.3, smooth_shading = False,
               diffuse = 0.8, specular = 0.5, specular_power = 2)
```



Podemos usar também os mapas de cores para representar as superfícies de terrenos com os comandos de iluminação e de materiais da biblioteca VTK com pyvista.

```
import pyvista
import pyvista as pv

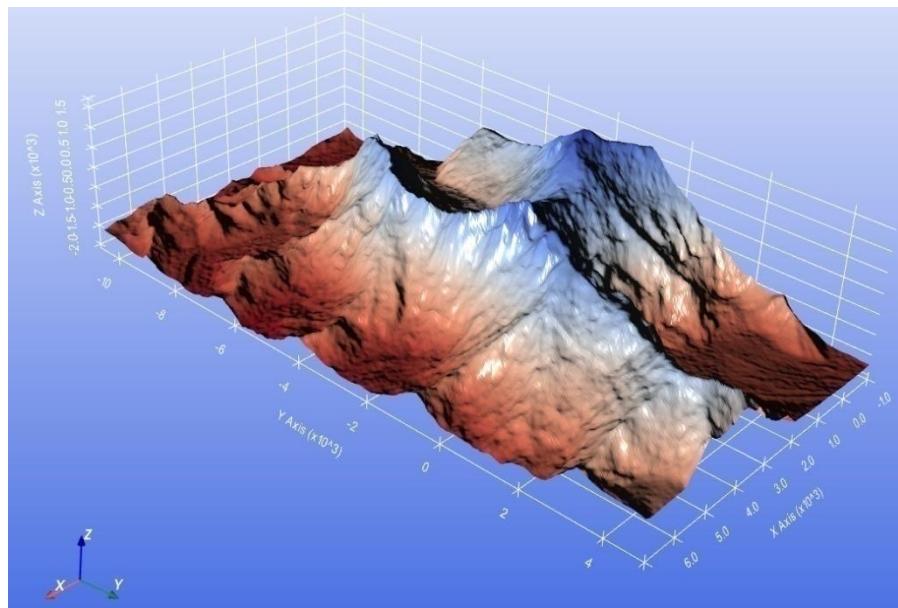
filename = 'C:/dados/everest.obj'
reader = pyvista.get_reader(filename)
mesh = reader.read()

p = pv.Plotter(lighting = 'none', window_size = [1000, 1000])
p.show_grid()
p.show_axes()

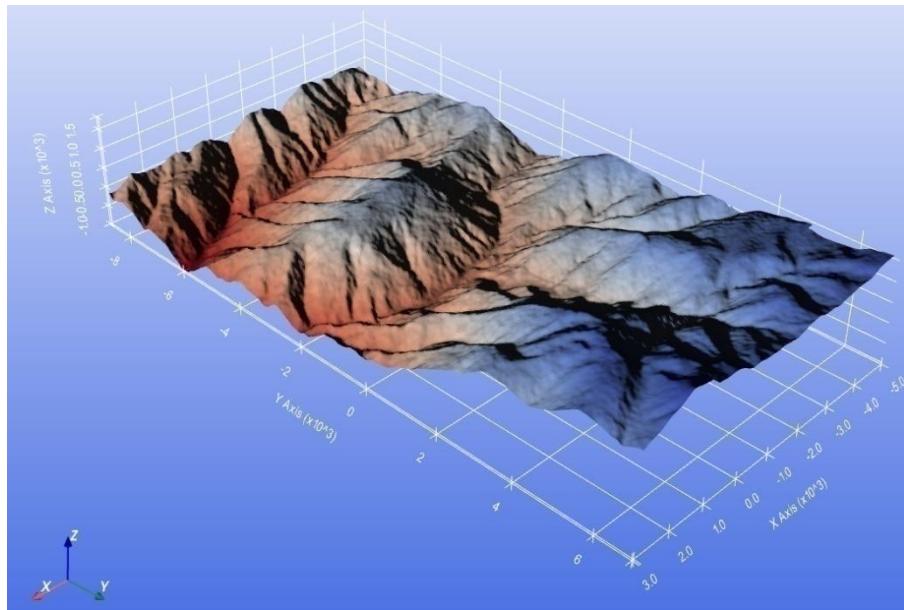
light = pv.Light(position = (10, 1, 1), light_type = 'scene light', intensity = 32)
p.add_light(light)

p.set_background('royalblue', top = 'white')
p.add_mesh(mesh, cmap = 'coolwarm_r', scalars = mesh.points[:, 2], show_scalar_bar = False,
           ambient = 0.3, diffuse = 0.5, specular = 0.5, specular_power = 15, pbr = True,
           metallic = 0.5, roughness = 0.2)

p.show()
```



```
filename = 'C:/dados/palcoyo.obj'
```



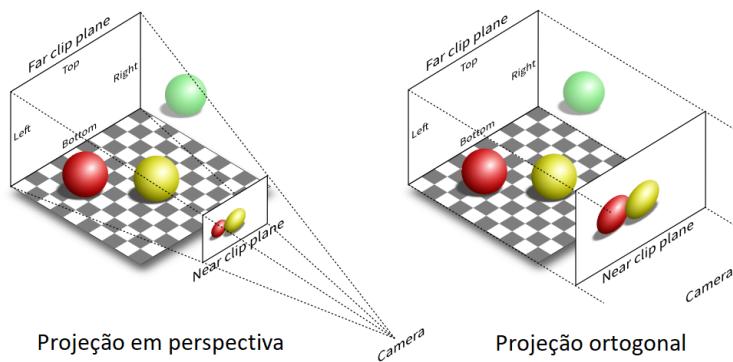
- Atividade 6**
- 6.1. Represente os objetos da atividade 5.5 com o uso da biblioteca PyVista.
 - 6.2. Utilize as funções de iluminação nos objetos representados da Atividade 6.1.
 - 6.3. Escolha um conjunto de coordenadas de uma superfície para representá-la com as funções de iluminação e de materiais mostrados nesta seção.
 - 6.4. Escolha um conjunto de coordenadas de um objeto para representá-lo com as funções de iluminação e de materiais mostrados nesta seção.

7. Câmera

A classe de câmera da biblioteca **Pyvista** adiciona um conjunto padrão de uma câmera que funciona em praticamente qualquer projeto de representação gráfica 3D. Porém, em alguns projetos precisamos modificar os valores dos atributos para melhorar a visualização dos observadores.

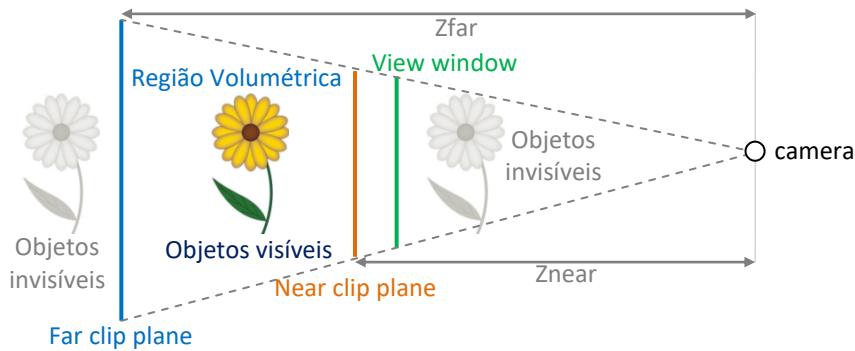
Uma câmera em Computação Gráfica cria o chamado *frustum* (tronco), que é determinado por uma pirâmide com vértice na lente da câmera no caso da projeção em perspectiva ou um prisma quando a projeção for

ortogonal. Nestes dois tipos de projeção temos 3 planos paralelos definidos: **Near clip plane**, **Far clip plane** e **View window**. A região do tronco de pirâmide ou do prisma entre os planos near clip plane e far clip plane define a região volumétrica que será visualizada pelo observador.

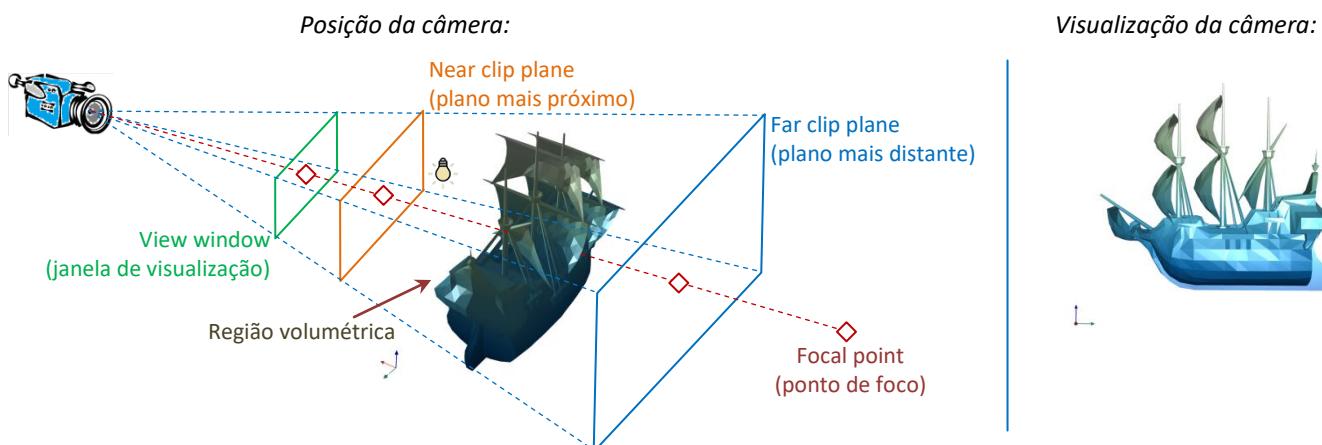


Disponível em: https://www.vhv.rs/viewpic/iwhJhTo_orthographic-projection-camera-hd-png-download/

Observando a vista lateral da projeção em perspectiva, conseguimos definir os valores das distâncias **Zfar** (entre a câmera e o far clip plane) e **Znear** (entre a câmera e o near clip plane). Estas medidas definem as regiões dos objetos visíveis e invisíveis da cena, e a diferença entre estas distâncias define o **clipping range**.

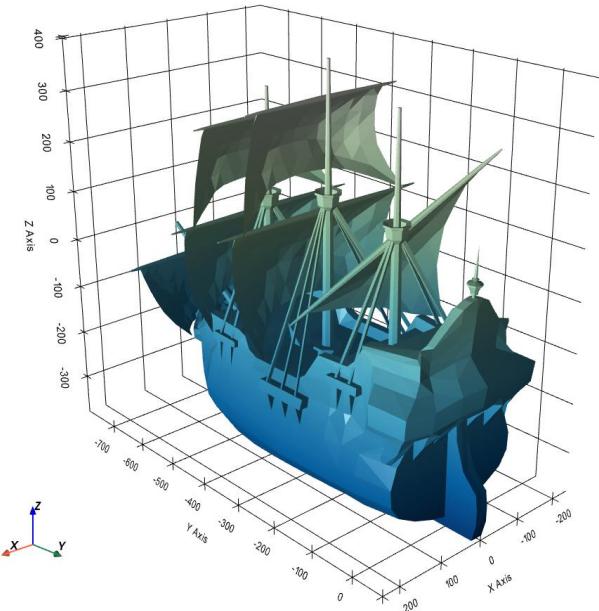
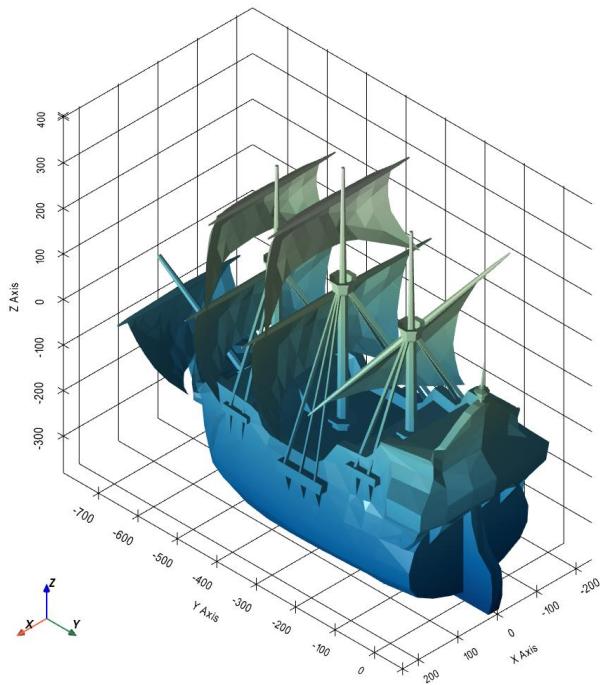


Quando representamos os objetos de uma cena por meio da projeção em perspectiva, as projeções da imagem da câmera nos planos near clip plane e view window ficam reduzidas. Entretanto, quando os objetos de uma cena são representados utilizando a projeção ortogonal, as medidas não se alteram.



O comando `enable_parallel_projection()` ativa a projeção paralela. Veja a diferença entre as visualizações paralela e em perspectiva mostradas a seguir.

```
p = pv.Plotter(lightning = 'none', window_size = [1000, 1000])
p.enable_parallel_projection()
```



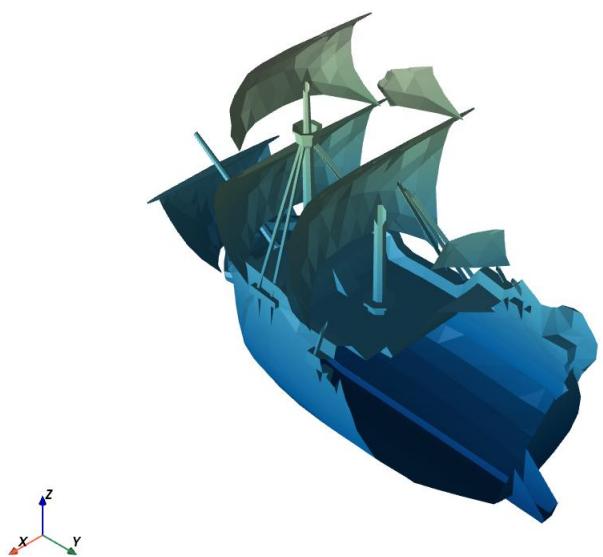
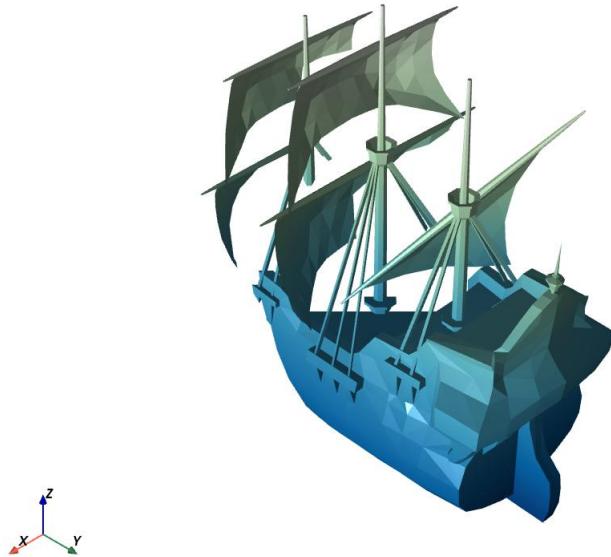
Podemos definir o zoom da camera com o comando mostrado a seguir. Na representação em perspectiva, o ângulo de visualização da câmera é ajustado pelo fator especificado. Quando usamos a representação ortogonal, a escala paralela é ajustada pelo fator especificado. Quando o valor do fator for maior do que 1, temos o **zoom-in** (aproximação); quando o valor deste fator for menor que 1, temos o **zoom-out** (afastamento).

Outro atributo que podemos verificar ou ajustar é o **clip plane**, que define as distâncias destes planos **Zfar – Znear**. Se os objetos estiverem fora da região volumétrica, o objeto pode aparecer cortado ou nem aparecer na janela de visualização.

```
p.camera.zoom(0.8)
print(p.camera.clipping_range)
```

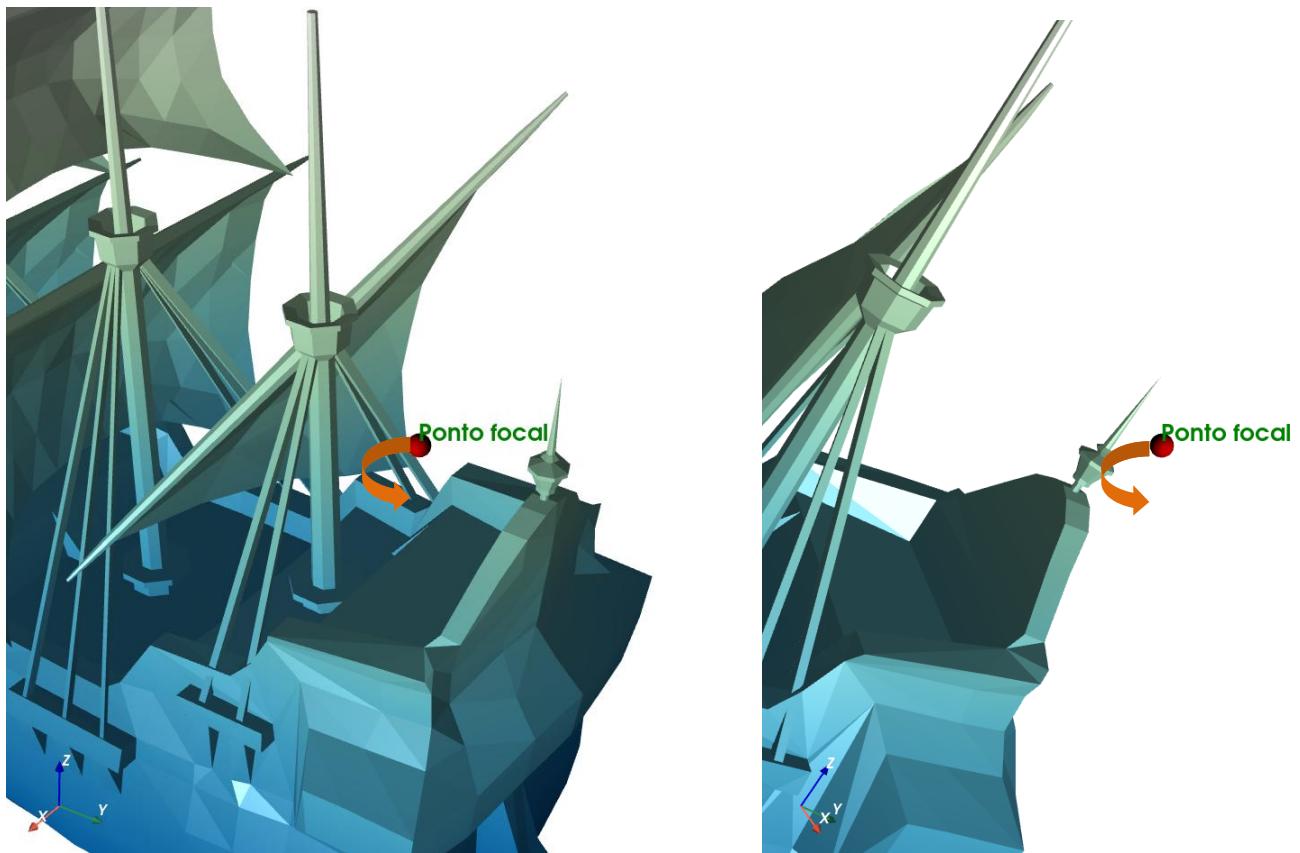
Se ajustarmos os valores dos planos com as seguintes medidas, temos as visualizações cortadas do objeto:

```
p.camera.clipping_range = (1000, 2500)
p.camera.clipping_range = (2400, 3500)
```



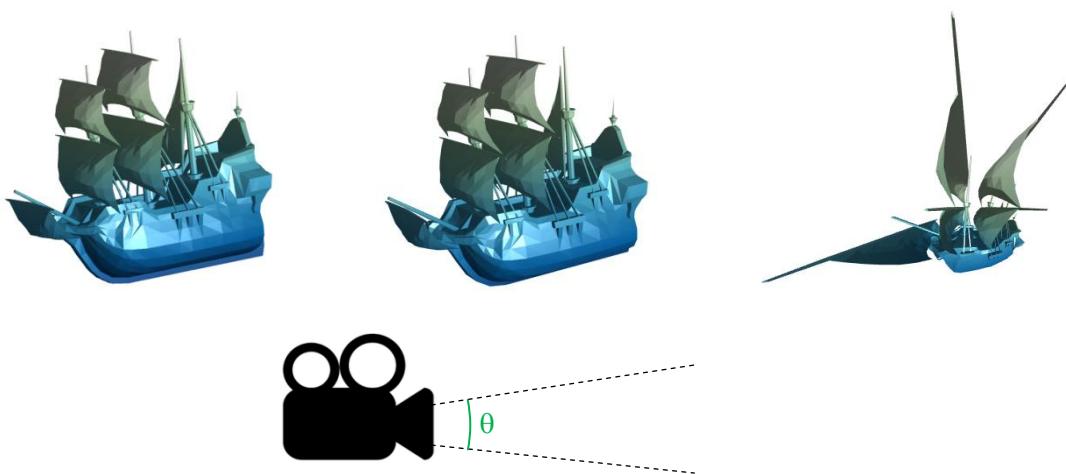
O ponto de foco define o centro de rotação da janela de visualização da cena. Quando não especificamos este ponto, o sistema VTK utiliza o centro do objeto representado, ou o centróide quando vários objetos estão na mesma cena. Se utilizarmos o ponto focal (100, 0, 150), o objeto será movimentado em torno deste ponto:

```
p.camera.focal_point = (100, 0, 150)
```



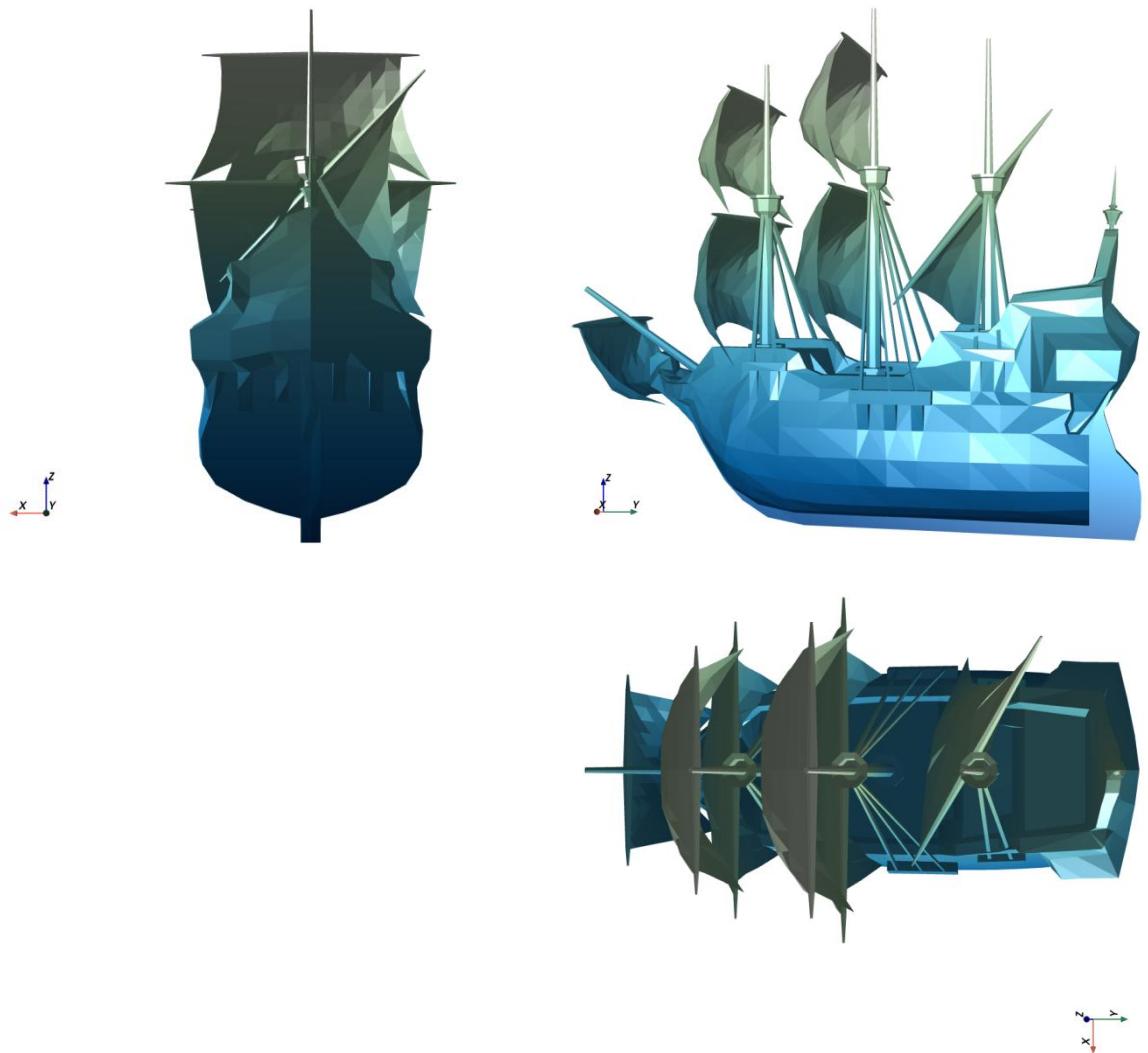
O ângulo de visualização θ determina a distorção da perspectiva. Quando temos ângulo θ próximo de zero, temos a perspectiva sem distorções, próxima à perspectiva paralela; porém, se o ângulo θ tem valor próximo de 180° , a perspectiva fica com distorções como se o objeto ficasse muito próximo do observador. Veja os exemplos com ângulos de 10° , 30° e 170° .

```
p.camera.view_angle = 10.0
p.camera.view_angle = 30.0
p.camera.view_angle = 170.0
```

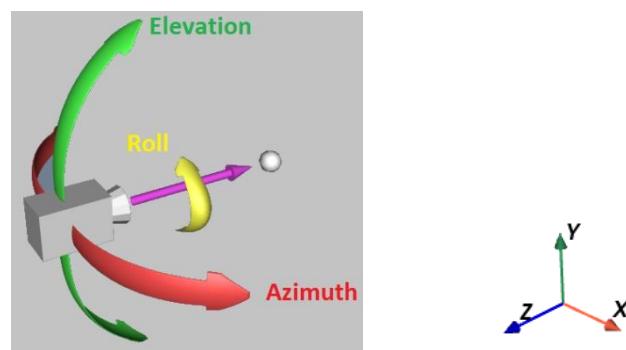


A posição da câmera pode ser definida usando o atributo `position`. No exemplo mostrado a seguir, podemos zerar uma das coordenadas para ter a visualização da cena como vista frontal, lateral ou superior. Por meio das coordenadas, podemos deixar a câmera mais próxima ou mais afastada dos objetos da cena.

```
p.camera.position = (1700, 0, 0)
p.camera.position = (0, 1700, 0)
p.camera.position = (0, 0, 1700)
```

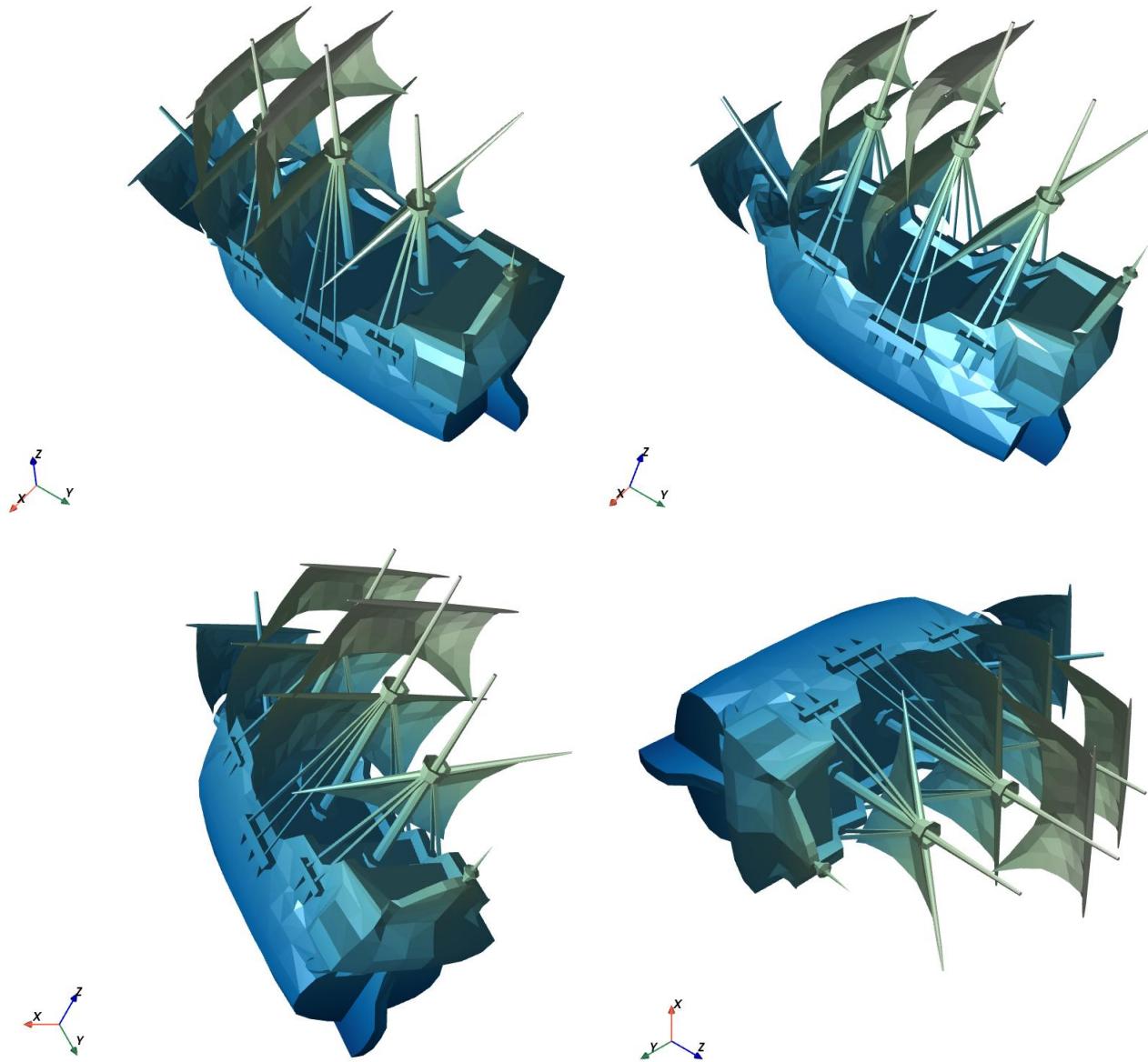


Além da posição da câmera, podemos modificar os ângulos `elevation`, `azimuth` e `roll`. Como padrão, estes ângulos possuem as seguintes medidas iniciais: `elevation` = 0, `azimuth` = 0 e `roll` = -120. Este valor modificado do atributo `roll` mostra os valores de coordenadas Z na vertical; porém, este eixo na computação gráfica é usado perpendicular à janela de visualização.



No primeiro exemplo, temos a mudança do ângulo `elevation`; no segundo exemplo, modificamos o ângulo `azimuth`; no terceiro e quarto exemplos modificamos as medidas do ângulo `roll`.

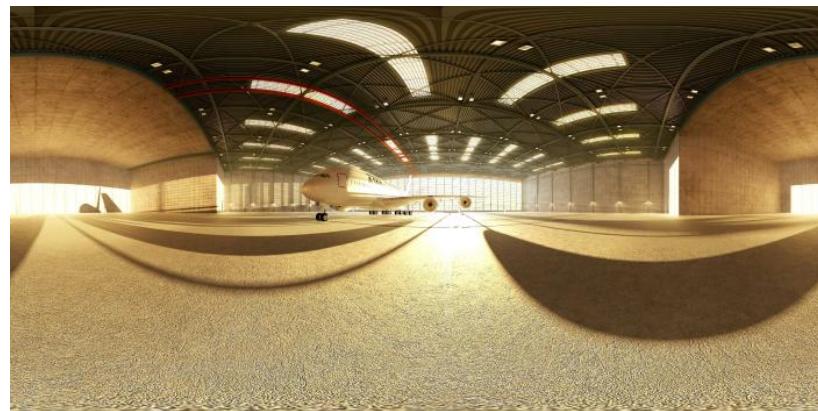
```
p.camera.elevation = 15
p.camera.azimuth = -30
p.camera.roll = -150
p.camera.roll = -240
```



Para deixar os ambientes realistas com efeitos convincentes em tempo real em jogos, podemos utilizar imagens de fundo (backgrounds) que transmitem uma visão muito ampla ou inteira em uma cena. Estas imagens são chamadas de esféricas, e possuem dois formatos muito utilizados: panorama equirretangular e cúbico. Como resultado, o observador experimenta a ilusão de estar cercado pela cena que foi usada para gerar o plano de fundo.

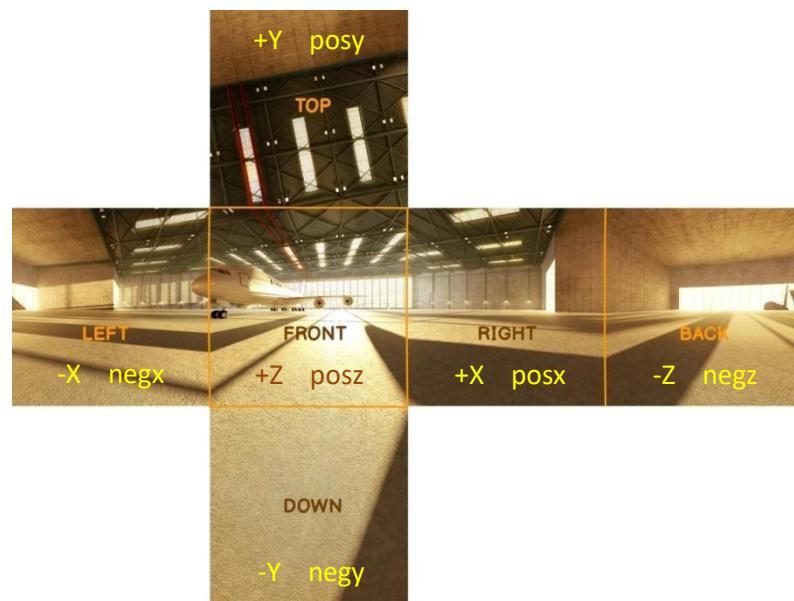
Utilizamos as projeções panorâmicas para mapear uma cena 3D total ou parcialmente em uma superfície bidimensional. Por exemplo, projeções cilíndricas transmitem a cena visível em todas as direções, exceto logo acima da cabeça e sob os pés do observador. Desta forma as partes superior e inferior do cilindro imaginário ficam com partes do ambiente não mapeadas quando são projetadas.

As imagens panorâmicas esféricas utilizam ângulos de visão vertical de 180° e horizontal de 360° . Estas imagens contêm pontos provenientes de todas as direções e, portanto, podem ser visualizadas quando estabelecemos uma relação de cada ponto da imagem com o respectivo ponto de uma esfera. Esse formato é bastante popular em mídias sociais e têm aplicações em programas gráficos 3D, simulações de interiores e jogos imersivos de computador. O denominado panorama equirretangular consiste em uma única imagem retangular cuja largura e altura têm razão 2:1. Estas imagens podem ser capturadas com uma câmera de 360° .

Imagen de fondo no ambiente 3D**Imagen equirretangular**

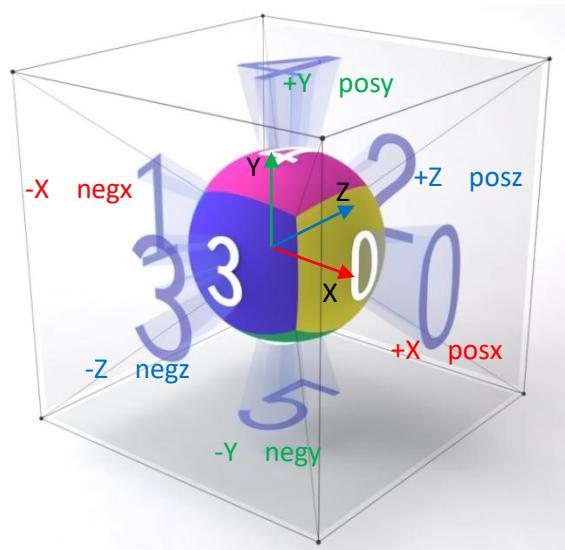
Disponível em: <https://onix-systems.com/blog/how-to-use-360-equirectangular-panoramas-for-greater-realism-in-games>

O formato cúbico, também chamado de cubemap, utiliza as faces de um cubo para preencher toda a esfera ao redor do visualizador. Esses mapas geralmente são criados pela imagem da cena com seis câmeras que possuem ângulos de visão de 90°, resultando nas seguintes texturas: esquerda, frontal, direita, traseira, superior e inferior. As seis imagens normalmente são organizadas como um cubo planificado. Usando o sistema de coordenadas do VTK, apenas a direção do eixo Z é invertida para nomenclatura das faces frontal e traseira.



Disponível em: <https://onix-systems.com/blog/how-to-use-360-equirectangular-panoramas-for-greater-realism-in-games>

Cada uma das faces do cubo tem seu respectivo mapa de textura. Quando dobradas, a visualização é remapeada para as faces de um cubo que se encaixam perfeitamente. Este tipo de mapeamento é chamado de **skybox**. Por um lado, uma imagem cúbica é um tipo de textura de superfície, ou seja, uma textura 2D comum. Por outro lado, é volumétrico porque cada ponto dentro do espaço de coordenadas da textura 3D corresponde a uma das faces do cubo.



Disponível em: <https://onix-systems.com/blog/how-to-use-360-equirectangular-panoramas-for-greater-realism-in-games>

O formato cúbico tem uma aplicação interessante na computação gráfica: o mapeamento de ambiente projetado na esfera pode ser usado para fazer os objetos parecerem brilhantes ou refletivos. Nestes casos, a textura do mapa deve ser uma visão da cena refletida em objetos com material metálico.

O formato cúbico permite que os desenvolvedores e designers de games criem ambientes ricos e inexploráveis com baixos custos de desempenho. Temos fotografias, imagens desenhadas à mão ou geometria 3D que são usadas como planos de fundo ou objetos inacessíveis. O renderizador gráfico cria as imagens como faces de um cubo a uma distância praticamente infinita do ponto de vista localizado no centro do cubo.

Com o sistema VTK/pyvista, podemos usar o mapeamento cúbico (cubemap) com alguns cuidados. Devemos rotacionar o objeto em torno do eixo x com ângulo de 90°, pois o sistema de coordenadas do skybox é diferente do sistema mostrado em outros exemplos que vimos anteriormente. Outro cuidado que devemos ter é de usar o atributo roll da câmera com ângulo 0, mantendo o eixo y na vertical.

Os arquivos que contém as imagens cubemap devem ter os nomes **negx**, **negy**, **negz**, **posx**, **posy** e **posz**. Indicando o diretório onde estão salvas estas imagens, usamos a função `cubemap.to_skybox()` como um ator na cena, e criamos o fundo da cena com a função `set_environment_texture`.

```
import pyvista
import pyvista as pv

filename = 'C:/dados/galleon.ply'
reader = pyvista.get_reader(filename)
mesh = reader.read()
mesh.rotate_x(-90.0)

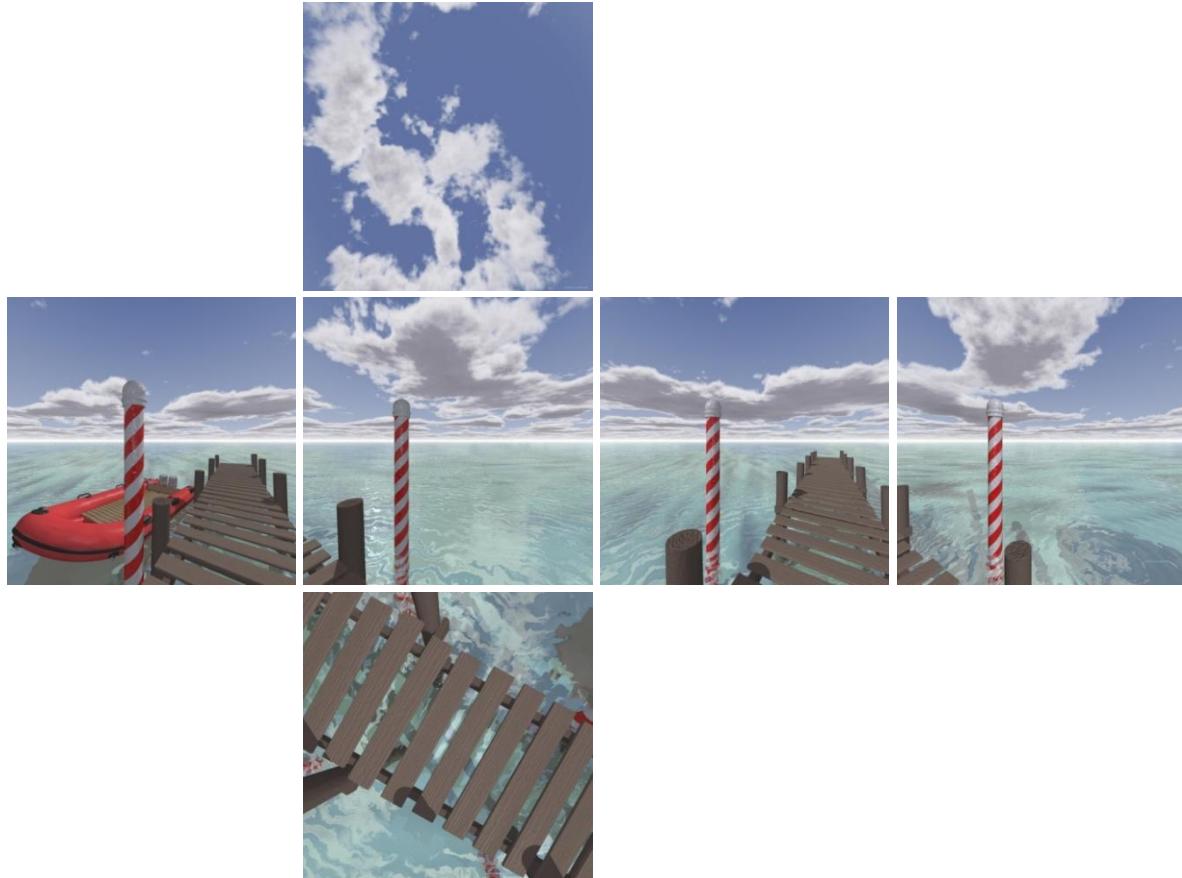
p = pv.Plotter()
p.show_axes()
light = pv.Light(position = (-10, 1, 1), light_type = 'scene light')
p.add_light(light)

cubemap = pyvista.cubemap('C:/dados/cubemap4')
p.add_actor(cubemap.to_skybox())
p.set_environment_texture(cubemap)

p.add_mesh(mesh, cmap = 'GnBu_r', scalars = mesh.points[:, 1], show_scalar_bar = False,
```

```
diffuse = 0.9, pbr = True, metallic = 0.8, roughness = 0.1)
```

```
p.add_axes()  
p.camera.roll=0  
p.show()
```



Disponível em: http://www.f-lohmueller.de/pov_tut/skyboxer/skyboxer_3.htm

Com as imagens no formato cubemap, temos o arquivo PLY com reflexos das texturas renderizadas.



Neste outro exemplo, usamos os atributos e efeitos para criar outro skybox para o objeto chopper.ply.



Disponível em: <https://www.humus.name/index.php?page=Textures>

Com estas novas imagens no formato cubemap, temos o arquivo PLY com reflexos das texturas renderizadas.

**Atividade 7**

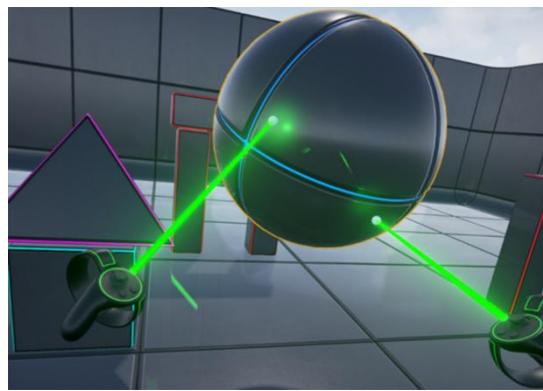
- 7.1. Modifique as configurações de câmera para as representações dos objetos da atividade 6.1.
- 7.2. Escolha uma imagem em formato cubemap e um objeto representado com a biblioteca PyVista para criar a cena com a imagem cubemap de fundo.
- 7.3. Escolha uma imagem em formato cubemap e uma superfície 3D representada com a biblioteca PyVista para criar a cena com a imagem cubemap de fundo.

8. Realidade Virtual

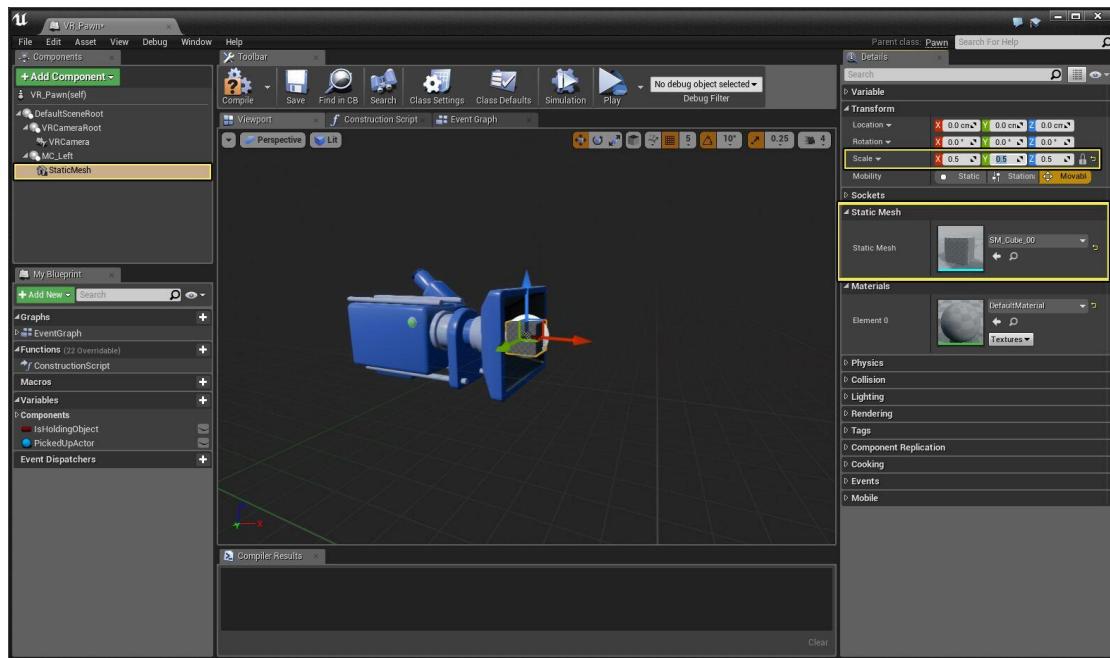
Em um ambiente programado com a tecnologia de Realidade Virtual (RV), ocorre uma simulação dos objetos em 3D, proporcionando ao visitante a sensação de que os objetos programados são reais. A Realidade Virtual cria um ambiente imersivo com manipulação dos objetos por meio de controles e óculos imersivos (MORO et al., 2017). Este tipo de imersão do usuário em um ambiente de Realidade Virtual pode ajudar na visualização de fenômenos físicos ou biológicos, simulações de situações de treinamentos, jogos educacionais, simulações de construções e outras áreas ligadas à educação.

O usuário tem uma percepção bastante realista quando entra em um ambiente programado de Realidade Virtual, podendo até incluir sensações tátteis na imersão. Nos últimos anos, esta área tem crescido bastante com a criação de óculos especiais e headsets que permitem que o usuário tenha a sensação bastante precisa do ambiente programado, seja para lazer (jogos), educação ou no ambiente empresarial.

Um dos softwares mais utilizados para criar ambientes de Realidade Virtual é o Unreal Engine, desenvolvido pela Epic Games. Este software é programado em C++, com suporte para as plataformas Windows, Linux, Mac OS e Mac OS X e diversos consoles de games.



Fonte: <https://docs.unrealengine.com/5.0/en-US/BuildingWorlds/VRMode/>



Fonte: <https://docs.unrealengine.com/5.0/en-US/motion-controller-component-setup-in-unreal-engine/>

Além das linguagens conhecidas de programação como Java e VTK, podemos programar os ambientes de Realidade Virtual usando páginas da web. A linguagem que vamos utilizar nesta seção chama-se A-frame, e está escrita nas linguagens HTML e Java. Trata-se de uma linguagem conhecida como WebXR, que possui comandos simples, é muito versátil, pode ser acessada em qualquer dispositivo com acesso à internet e cria experiências imersivas em 3D, Realidade Virtual e Realidade Aumentada em seu navegador.

O A-Frame usa a API WebXR para obter acesso aos dados do sensor do headset VR (posição, orientação) para transformar a câmera e renderizar conteúdo diretamente para headsets VR. Mas afinal, qual é a diferença entre WebVR e WebXR? A WebVR foi considerada uma API experimental projetada para ajudar os escritores de especificações a determinar as melhores abordagens para criar uma API de realidade virtual na Web. Depois de alguns anos de experimentos, os desenvolvedores fizeram algumas correções, dando origem à chamada WebXR. A diferença fundamental é que a WebXR suporta não apenas a Realidade Virtual, mas também a Realidade Aumentada, que combina objetos virtuais com o ambiente do usuário.

Outra diferença importante é que o WebXR tem suporte integrado para os controladores de entrada avançados que são usados com a maioria dos fones de ouvido de realidade mista, enquanto o WebVR contava com a API do Gamepad (periférico de entrada mais comum nos consoles de jogos modernos) para dar suporte aos controladores. No WebXR, as ações primárias de seleção e compressão são diretamente suportadas por meio de eventos, enquanto outros controles estão disponíveis por meio de uma implementação especial específica do WebXR do objeto Gamepad.

As tags a-frame são usadas para criar aplicações em 360° em navegadores de internet. Sua estrutura é praticamente a mesma das tags HTML, com possibilidade de personalizar a aplicação com a linguagem CSS. Ela funciona com os scripts **Java three.js**. A tag estrutural que define a cena contém todos os elementos 3D, como objetos, luzes, câmera, áudio, vídeo e permitem o uso dos dispositivos de imersão em Realidade Virtual, tais como HTC Vive, Oculus Rift, Samsung GearVR, smartphones e Google Cardboard. A documentação usada neste capítulo está baseada em no material disponível em <https://aframe.io/>

As principais tags estruturais de uma cena programada em a-frame estão descritas a seguir.

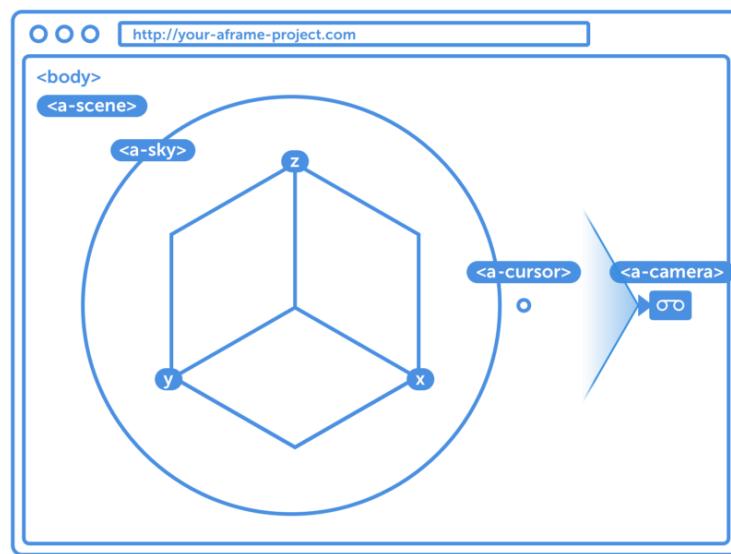
<a-scene> é a tag principal do projeto, que consiste no cenário da cena. Ela deve englobar todas as tags e inicia a programação 3D do ambiente. As referências de coordenadas X, Y e Z devem ficar dentro desta tag.

<a-assets> contém todos os recursos de áudio, vídeo, componentes, imagens e texturas da cena. É a primeira tag carregada na cena.

<a-cursor> é uma tag opcional que indica o cursor (mira no centro da tela). Com esta opção, podemos capturar o clique na cena.

<a-sky> é a definição do “céu” do ambiente, ou seja, a imagem de fundo da cena.

<a-camera> define as configurações da câmera usada pelo usuário. Sem esta tag definida, o usuário não consegue mudar o ponto de vista e interagir com objetos na cena. Podemos configurar mais de uma câmera na cena, principalmente quando temos ambientes de jogos.



Fonte: <https://tableless.com.br/realidade-virtual-com-a-frame/>

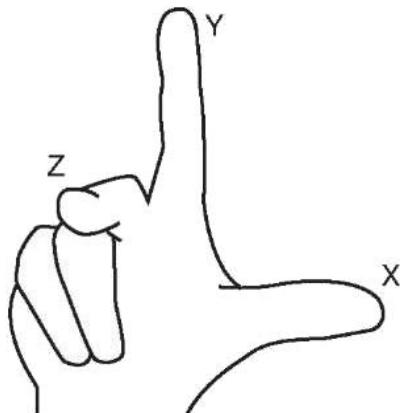
O framework que permite a renderização do ambiente é o **aframe.min.js**. Outras referências já programadas em JavaScript podem ser inseridas no cabeçalho da página e colocadas como scripts da seguinte forma:

```
<script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
```

Logo, a estrutura HTML do a-frame é a seguinte:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-assets>
```

```
        </a-assets>
        <a-sky> </a-sky>
        <a-camera> </a-camera>
        <a-cursor> </a-cursor>
    </a-scene>
  </body>
</html>
```



O sistema de coordenadas é o mesmo que usamos em Computação Gráfica com as bibliotecas VTK e PyVista. O eixo z permanece como a coordenada que “sai” da tela, e a unidade linear do a-frame é o metro, enquanto que a unidade angular é o grau.

O posicionamento de componentes (objetos e câmeras) é feito com a propriedade **position**. As transformações geométricas de escala (**scale**) e rotação (**rotation**) podem ser feitas diretamente nas tags dos componentes.

Outras propriedades comuns de HTML são usadas para definir os componentes das cenas, tais como **width** (largura), **height** (altura) e **color** (cor).

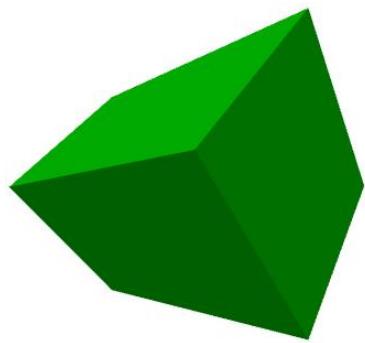
Vamos criar uma cena com as tags básicas do a-frame.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-box color="green"></a-box>
    </a-scene>
  </body>
</html>
```

Você consegue visualizar este cubo no navegador de internet? Como este cubo não tem posição definida, ele está com coordenadas (0, 0, 0) e dimensões de largura, altura e profundidade iguais a 1. A câmera também tem estas coordenadas, caso não seja definida. Logo, para visualizar este cubo, podemos colocar uma coordenada z negativa na posição do objeto, ou definir uma câmera com coordenada z positiva:

```
<a-scene>
  <a-box color="green" position="0 2 -4"></a-box>
</a-scene>
```

Para dimensionar o cubo, podemos usar a função **scale** ou as dimensões **width**, **height** e **depth**. A rotação pode ser feita em torno dos eixos x, y e z, respectivamente. Por exemplo, para dimensionar nosso cubo com largura e altura iguais a 2, profundidade 3 e rotacioná-lo segundo 45° em torno dos eixos y e z, basta usar a tag a seguir.



```

<a-scene>
  <a-box color="green" position="0 2 -4" rotation="0 45 45" width="2" depth="3"
    height="2"></a-box>
</a-scene>
ou
<a-scene>
  <a-box color="green" position="0 2 -4"
    rotation="0 45 45" scale="2 2 3"></a-box>
</a-scene>

```

Ao clicar no símbolo **VR** no canto inferior direito da tela, temos a experiência imersiva, que pode ser visualizada com óculos de Realidade Virtual ou então manipulada com os controles do teclado e mouse do computador. Os controles do teclado podem ser as setas, o click do mouse e as teclas **WASD**.

Para criar o fundo da cena, definimos o céu (**sky**) utilizando cores ou uma imagem 360°, também conhecida por equirectangular, projeção cilíndrica equidistante ou projeção geográfica.

Para criar o fundo da página somente com cor, usamos a tag:

```
<a-sky color="#99ccff"></a-sky>
```

Um tipo de fundo que podemos utilizar no ambiente de Realidade Virtual é um pré-definido com componente **JavaScript**. Para inserir um destes fundos, primeiro vamos colocar a tag de script no cabeçalho da página.

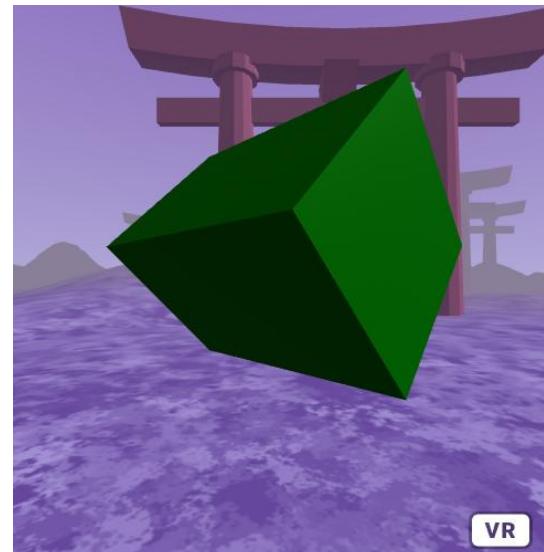
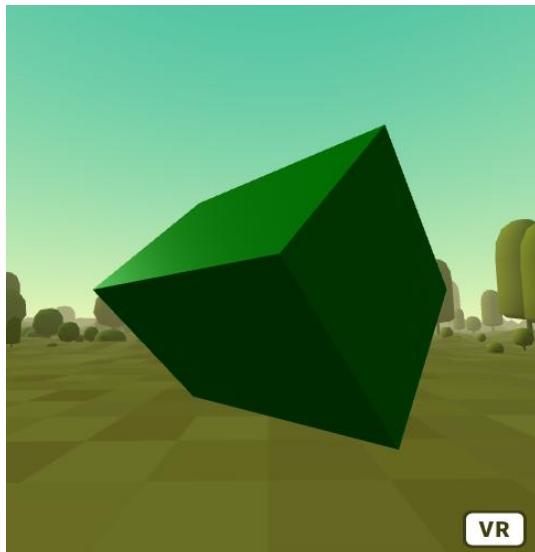
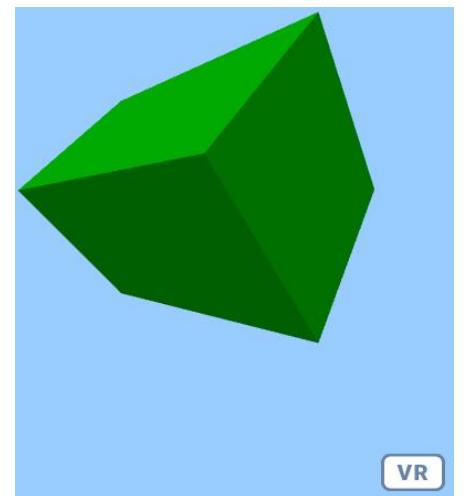
O link do JavaScript pode ser feito diretamente do site, ou você pode salvar o arquivo em uma pasta de scripts. Seguem abaixo as duas opções para criação de fundos pré-definidos.

```
<script src="https://unpkg.com/aframe-environment-
component/dist/aframe-environment-component.min.js"></script>
```

ou

```
<script src=".//scripts/aframe-environment-component.min.js"></script>
```

Depois de “linkar” o script do ambiente, podemos inserir a tag como **a-entity**. Esta tag serve para inserir muitos objetos dentro das nossas páginas de RV. Neste caso, a função é **environment**: a propriedade **preset** modifica o ambiente com as opções **none**, **default**, **contact**, **egypt**, **checkerboard**, **forest**, **goaland**, **yavapai**, **goldmine**, **threetowers**, **poison**, **arches**, **tron**, **japan**, **dream**, **starry** e **osiris**; **dressingAmount** é a quantidade de objetos na cena. No exemplo **forest**, colocamos 500 árvores na cena.



```
<a-entity environment="preset: forest; dressingAmount: 500"></a-entity>
```

As texturas e imagens 360° só podem ser visualizadas quando o arquivo HTML estiver publicado em um servidor HTTP para Realidade Virtual ou HTTPS para Realidade Aumentada. Para publicar a página e inserir uma imagem 360° no fundo, usamos as tags de **assets** para definir o caminho da imagem. Crie uma pasta chamada **imagens** no diretório do arquivo HTML do cubo e insira uma imagem em formato 360° nesta pasta. Vamos colocar o identificador **id** com o nome **ceu**.

Vamos construir um cilindro, diminuir as medidas do cubo e colocar texturas nos objetos da cena. Basta salvar os arquivos de imagem das texturas na pasta de **imagens**, identificá-las com **id** nos **assets** e ao invés de colocar a propriedade de cor, colocar a ligação de cada objeto com sua respectiva textura com a propriedade **src** (source).

```
<a-assets>
  
  
  
</a-assets>

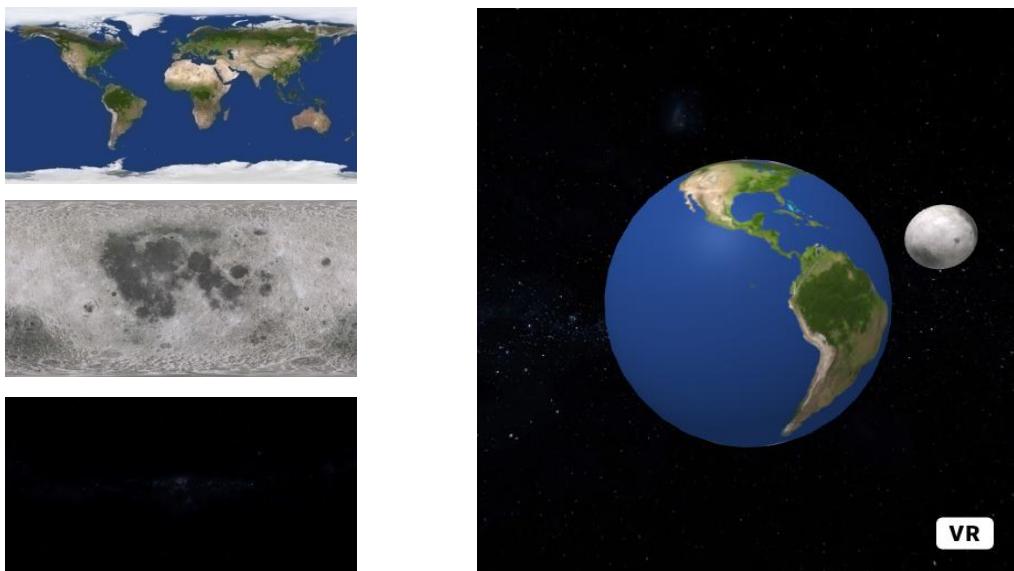
<a-box src="textura1" position="0 1 -4" rotation="0 45 45" scale="1 1.5 1"></a-box>
<a-cylinder src="textura2" position="2.5 1 -4" radius="0.5" height="2"></a-cylinder>
<a-sky src="#ceu"></a-sky>
```

O resultado é o ambiente imersível mostrado a seguir, com os sólidos representados. O cubo e o cilindro ficam fixos, e ao movimentarmos a tela (seja com os controles dos óculos ou teclado do computador), o visitante da página fica com a sensação de imersão total na cena. Adicione outros sólidos na cena, e modifique texturas e a imagem de fundo da cena. Os comandos para representar outros sólidos são os seguintes:

```
<a-sphere src="#textura" position="0 0 -5" radius="2"></a-sphere>
<a-plane src="#textura" position="0 0 -5" width="4" height="3"></a-plane>
<a-circle color="#FF926B" radius="20"></a-circle>
<a-cone color="tomato" radius-bottom="2" radius-top="0.5"></a-cone>
<a-dodecahedron color="orange" radius="5"></a-dodecahedron>
<a-icosahedron color="yellow" radius="5"></a-icosahedron>
<a-image src="#imagem1"></a-image>
<a-octahedron color="aliceblue" radius="5"></a-octahedron>
<a-ring color="#B84A39" radius-inner="1" radius-outer="2"></a-ring>
<a-tetrahedron color="green" radius="5"></a-tetrahedron>
<a-text value="Projeto de RV"></a-text>
<a-torus-knot color="#43A367" arc="180" p="2" q="7" radius="5"
  radius-tubular="0.1"></a-torus-knot>
<a-torus color="aqua" arc="270" radius="5" radius-tubular="0.1"></a-torus>
```



Podemos usar a mesma estrutura utilizando imagens equirectangulares como texturas de esferas que representam a Terra e a Lua, além da Via Láctea que pode ser usada como o céu da cena.

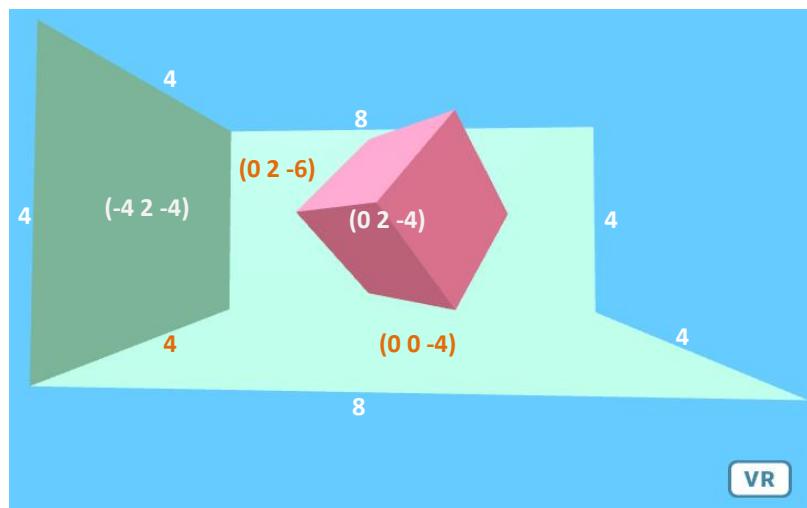


8.1. Iluminação

A iluminação em um ambiente programado com a-frame pode ser feita com as seguintes propriedades: `ambient`, `spot`, `point`, `hemisphere` e `directional`. Em todos os casos podemos definir as informações de posição, cor e intensidade do tipo de iluminação escolhido. Caso contrário, assume-se a posição $(0, 0, 0)$, cor branca e intensidade 1. Para estudar os efeitos de cada tipo de luz em objetos na cena, vamos criar 3 planos para representar um diedro, com larguras e alturas iguais a 8 e 4.

Afastando-se o primeiro plano com distância $z = -6$, temos o plano do fundo da cena; com distância $z = -4$ e rotação de -90° em torno do eixo x, temos o plano do piso da cena; o plano lateral pode ser construído afastando-se com coordenadas $x = -4$ e $z = -4$ e rotacionando-o em torno do eixo y com ângulo de 90° . Vamos colocar um box na cena, com mesmas coordenadas que usamos anteriormente.

```
<a-plane color="#A9F5D0" position="0 2 -6" width="8" height="4"></a-plane>
<a-plane color="#A9F5D0" position="0 0 -4" rotation="-90 0 0" width="8" height="4"></a-plane>
<a-plane color="#A9F5D0" position="-4 2 -4" rotation="0 90 0" width="4" height="4"></a-plane>
<a-box color="#F7819F" position="0 2 -4" rotation="0 45 45" scale="2 2 2"></a-box>
```

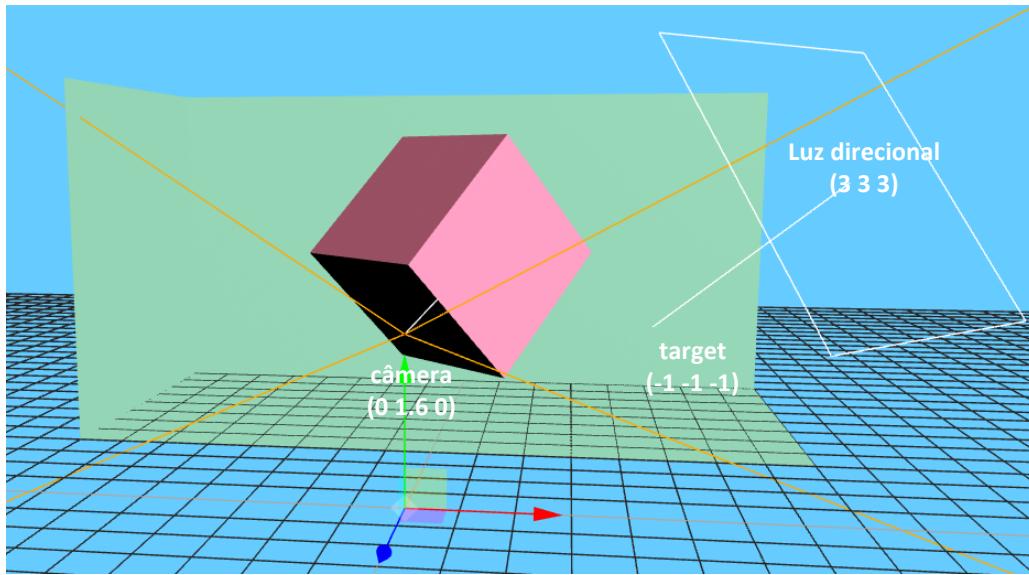


A **luz ambiente** afeta todos os elementos da cena com mesma intensidade. Pode ser usada como auxiliar para as demais luzes. Recomenda-se seu uso para áreas sombreadas. Usando-se a tag específica `light`, temos a luz ambiente de cor amarela e intensidade igual a 0.5 definida da seguinte forma:

```
<a-light type="ambient" color="yellow" intensity="0.5"></a-light>
```

Como se trata de um elemento da cena, temos que definir a luz fora dos assets. A luz ambiente não tem efeitos com rotações, escalas e posicionamentos, pois é um tipo de luz que afeta por igual todos os elementos da cena.

Os efeitos da **luz direcional** simulam uma fonte distante, como a luz do sol. O elemento mais importante deste tipo de luz é a distância: geralmente, podemos colocar a posição desta luz com mais de 100m, produzindo a simulação de sombras paralelas, como as sombras produzidas pelo sol.



Outro elemento importante da luz direcional é o alvo (target), que pode ser definido como uma tag filha da tag de luz. As coordenadas ficam sendo relativas à posição da luz. No exemplo mostrado a seguir temos a luz com posição (3, 3, 3) com tag filha direcional (-1, -1, -1) que simula a direção que a luz atinge os objetos na cena.

```
<a-light type="directional" intensity="1.5" position="3 3 3" target="#directionaltarget">
  <a-entity id="directionaltarget" position="-1 -1 -1"></a-entity>
</a-light>
```

Os efeitos de sombras podem ser produzidos inserindo-se a propriedade shadow. Esta propriedade é desabilitada por padrão, pois as sombras deixam o ambiente mais carregado e demorado para renderização da cena. A definição da sombra ocorre em dois elementos: o que produz a sombra, e o que recebe a sombra. Podemos também colocar a propriedade dupla em alguns objetos da cena, ou seja, eles podem produzir e receber efeitos de sombras. No exemplo dos planos, podemos colocar em cada tag a propriedade que eles recebem sombra:

```
shadow="receive: true"
```

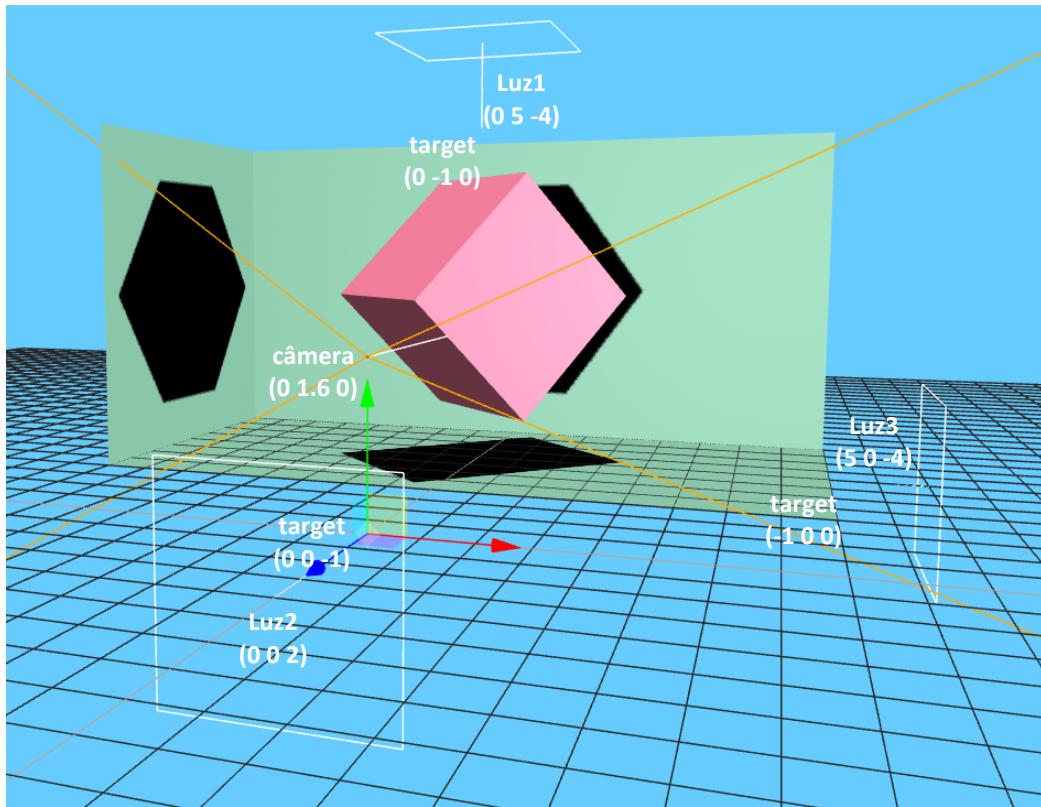
O elemento a-box vai produzir a sombra, ou seja, a tag a-box terá a seguinte propriedade adicional:

```
shadow="cast: true"
```

Podemos criar 3 luzes que simulam as projeções ortogonais das sombras dos objetos nos três planos. Para cada uma delas, temos que definir a produção de sombra, e targets diferentes. Assumindo-se uma distância de 5 metros de cada ponto de luz, temos o ambiente com sombras a seguir.

```
<a-light type="directional" intensity="0.8" position="0 5 -4" light="castShadow:true"
  target="#directionaltargetY">
  <a-entity id="directionaltargetY" position="0 -1 0"></a-entity>
</a-light>
<a-light type="directional" intensity="0.8" position="0 0 2" light="castShadow:true"
  target="#directionaltargetZ">
  <a-entity id="directionaltargetZ" position="0 0 -1"></a-entity>
</a-light>
```

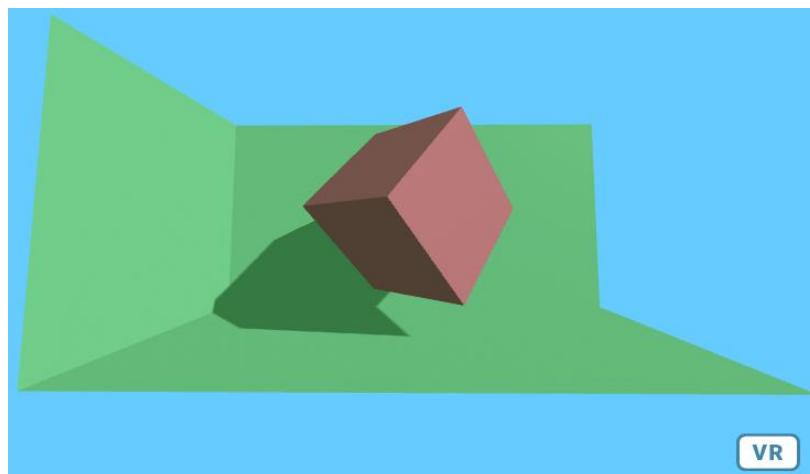
```
<a-light type="directional" intensity="0.8" position="5 0 -4" light="castShadow:true"
  target="#directionaltargetX">
  <a-entity id="directionaltargetX" position="-1 0 0"></a-entity>
</a-light>
```



O padrão de tamanho do quadrado desta fonte de luz é de 5 metros. Caso algum objeto tenha sombra cortada, podemos aumentar com as propriedades shadowCamera, colocadas na tag principal de cada luz. Se a sombra deste tipo de luz ficar com falhas de renderização, podemos adicionar a propriedade shadowCameraFar no atributo light, com valor maior do que 1.000.000.

```
light="castShadow:true; shadowCameraTop:10; shadowCameraRight:10; shadowCameraLeft: -10;
shadowCameraBottom:-10; shadowCameraFar: 1000000"
```

O tipo de luz **hemisphere** simula uniformemente duas cores de luzes: uma nos objetos e outra nos elementos de fundo da cena. É útil para simular efeitos de céu e gramados e funciona da mesma forma que a luz ambiente. Também é usada como luz de efeito auxiliar, pois não tem um ponto específico e não produz sombra. A tag desta luz define a cor principal e a cor de fundogroundColor. Usando-se esta luz junto com a luz direcional, podemos dar um tom mais esverdeado nos planos, colocando menos intensidade nos dois tipos de luzes.



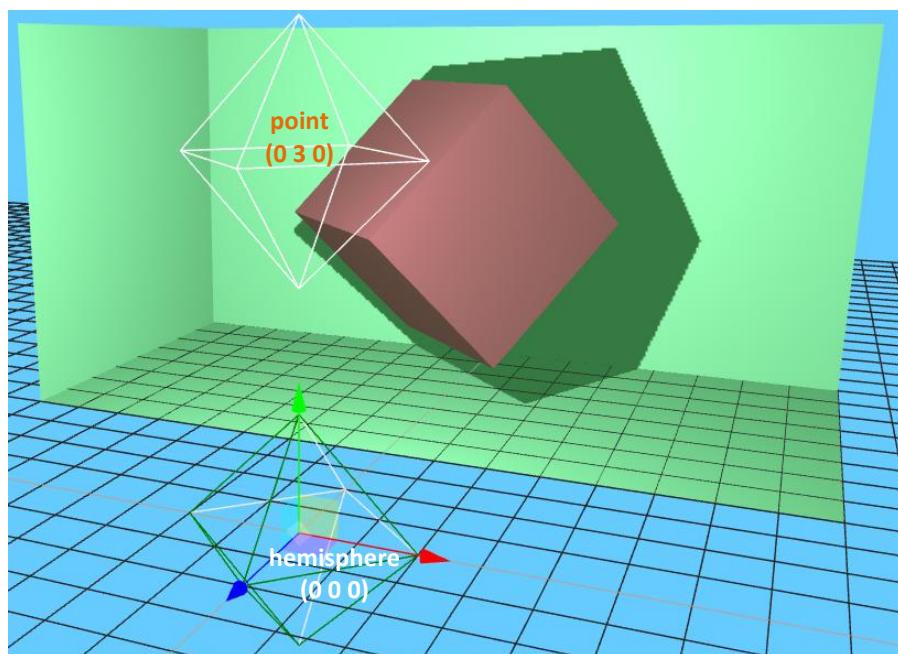
```
<a-light type="hemisphere" color="#eaeaea" light="groundColor:green" intensity="0.7"></a-light>
<a-light type="directional" intensity="0.5" position="8 5 5" light="castShadow: true"
  target="#directionaltargetZ">
  <a-entity id="directionaltargetZ" position="-1.3 -1 -1"></a-entity>
</a-light>
```

O comando que usamos para renderizar as sombras com mais suavidade é colocado na tag principal a-scene.

```
<a-scene shadow="type:pcfsoft">
```

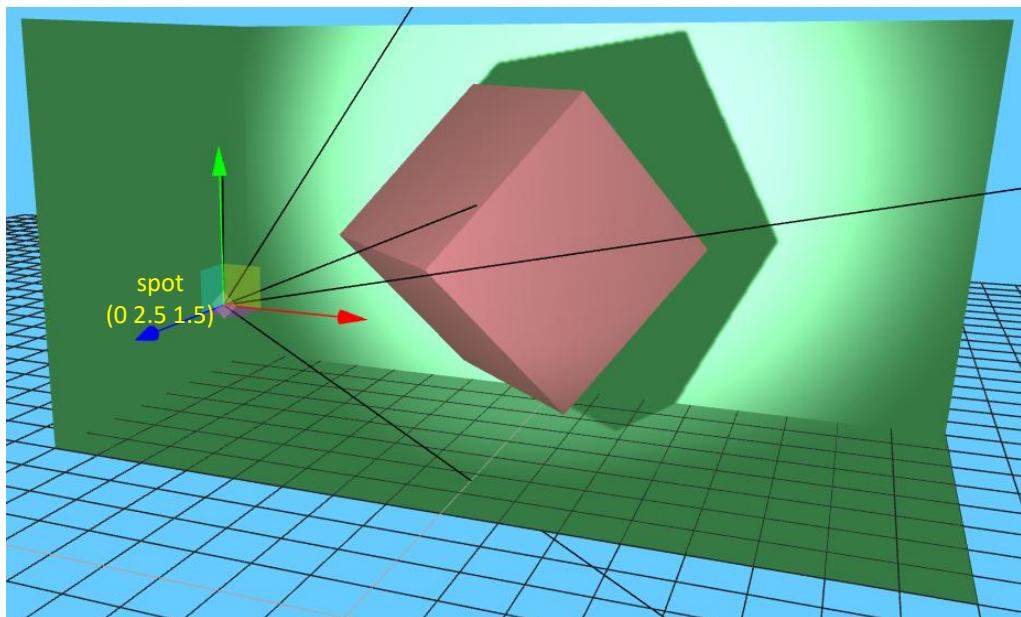
O tipo de luz **point** simula a iluminação de uma lâmpada na cena. Logo, a posição deste elemento é muito importante, pois afeta o tamanho da sombra de cada objeto. Outras propriedades que este tipo de luz tem são: decaimento (decay), que determina o quanto o efeito de luz decai em uma distância (distance) definida. No exemplo mostrado a seguir a luz tem decaimento 2 com a distância de alcance de 50 metros.

```
<a-light type="hemisphere" color="#eaeaea" light="groundColor:green" intensity="0.7"></a-light>
<a-light type="point" intensity="0.75" [distance="50" decay="2"] position="0 3 0"
  light="castShadow: true"></a-light>
```

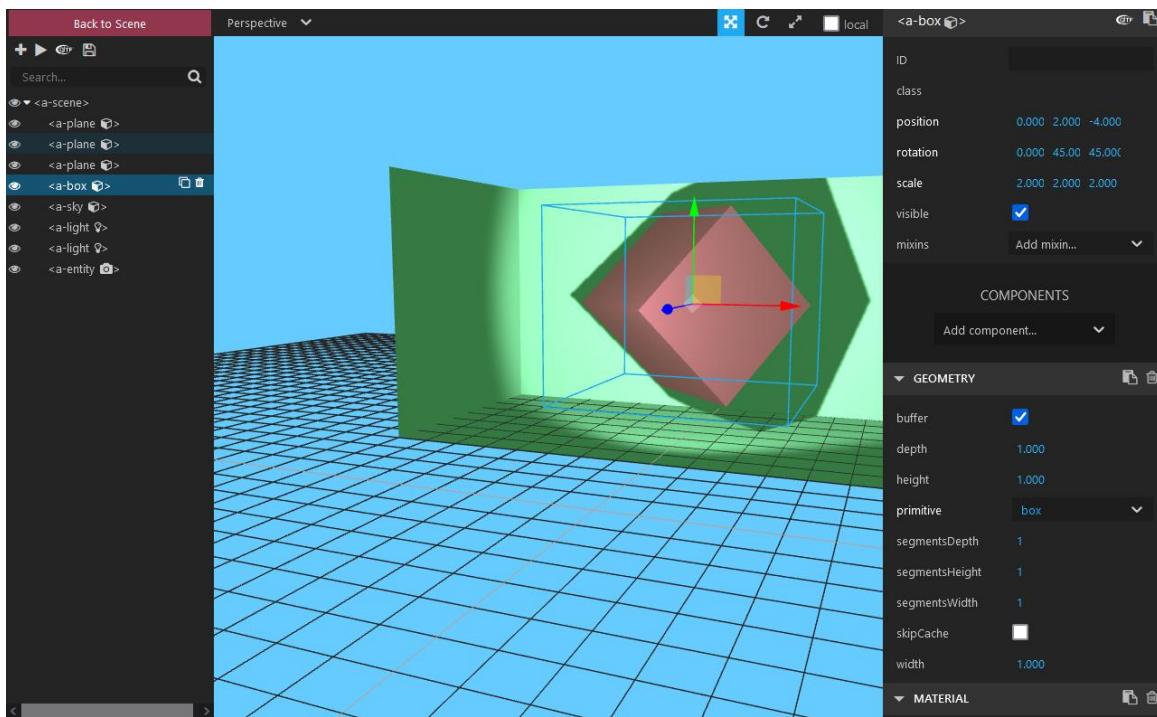


O tipo de iluminação **spot** simula uma lâmpada na cena, com o tipo de luminária em cone. As propriedades são similares ao tipo **point**, com definições de abertura (angle) da iluminação e do percentual de transição da parte iluminada para a parte escura (penumbra). O padrão desta iluminação é no sentido do eixo z, porém, podemos usar a transformação de rotação para mudar a direção do cone de iluminação. Usando a luz spot junto com a luz hemisphere temos a iluminação mostrada a seguir.

```
<a-light type="hemisphere" color="#eaeaea" light="groundColor:green" intensity="0.7"></a-light>
<a-light type="spot" intensity="0.75" [angle="30" penumbra="0.5" light="castShadow:true"
  position="0 2.5 1.5"></a-light>
```



A configuração de qualquer elemento da cena pode ser feita pressionando CTRL + ALT + i, que habilita o inspetor de cena. Após verificar os valores de novas configurações, você deve salvar um novo arquivo HTML para que as alterações tenham efeito. O inspetor só serve para visualizar a cena e cada um de seus elementos.



8.2. Animações

As animações podem ser inseridas dentro das tags dos elementos da cena. Por exemplo, para modificar o raio da esfera de 1.5 para 1.8, basta colocar a propriedade animation dentro da tag da esfera, indicando que o atributo com animação será o raio e o valor final será 1.8. Para o efeito de vai e vem, a direção é alternate (outros valores são: normal e reverse), a duração é em milisegundos (segundo/1000).

```
<a-sphere color="blue" position="0 1.5 -4" radius="1.5" animation="property: radius; to: 1.8; loop: true; dur: 5000;"></a-sphere>
```

Para deixar a animação com efeito linear e alternada, podemos inserir as seguintes propriedades de animação:

```
easing: linear; dir: alternate;
```

No exemplo da Terra e da Lua, podemos inserir iluminação do tipo spot criar as rotações das duas esferas em torno de seus próprios eixos por meio das seguintes tags:

```
<a-sphere src="#textura1" scale="2 2 2" animation="property: rotation; from: 0 0 0; to: 0 360 0; loop: true; dur: 10000; easing: linear;"></a-sphere>
<a-sphere src="#textura2" position="3 0 0" scale="0.5 0.5 0.5" animation="property: rotation; from: 0 360 0; to: 0 0 0; loop: true; dur: 5000; easing: linear;"></a-sphere>
```

Muitas vezes, precisamos que o elemento que será rotacionado possua uma referência diferente de seus próprios eixos. Para compreender a hierarquia usada no a-frame para criarmos as animações “aninhadas”, vamos construir os eixos x, y e z com cilindros usando tags mostradas a seguir.

O padrão da tag de cilindro é de construí-lo na direção do eixo y, com metade da altura para cima e metade para baixo do ponto de referência. Por isso, usamos position para deslocá-lo com 0.5m para a direita.

```
<a-cylinder radius="0.02" height="3" position="0 0.5 0" color="rgb(0,255,0)"></a-cylinder>
```

Para deixar os cilindros dos eixos paralelos aos eixos x e z, usamos as rotações de 90º em torno de z e x, respectivamente:

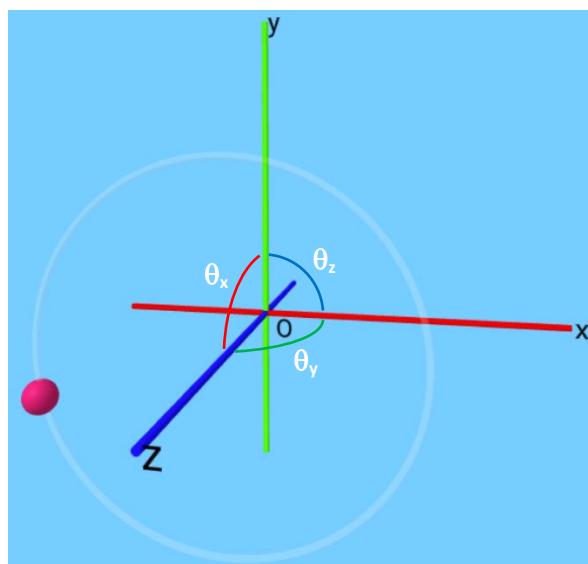
```
<a-cylinder rotation="0 0 90" radius="0.02" height="3" position="0.5 0 0" color="rgb(255,0,0)"></a-cylinder>
<a-cylinder rotation="90 0 0" radius="0.02" height="3" position="0 0 0.5" color="rgb(0,0,255)"></a-cylinder>
```

Vamos escrever os nomes dos objetos com tags de texto. Os tamanhos podem ser definidos pela propriedade width ou scale. Para deixar o nome da origem mais visível, podemos colocar um valor de -0.1 para a coordenada y, deixando-o um pouco abaixo da coordenada (0 0 0).

```
<a-text position="0.05 -0.1 0" value="O" width="4" color="black"></a-text>
<a-text position="2 0 0" value="x" width="4" color="black"></a-text>
<a-text position="0 2 0" value="y" width="4" color="black"></a-text>
<a-text position="0 0 2" value="z" width="4" color="black"></a-text>
```

Vamos construir uma esfera que terá efeito de animação de rotação em torno do eixo z. Primeiro, criamos uma entity com a animação, e dentro desta tag entity podemos indicar a posição da esfera.

```
<a-entity animation="property: rotation; to: 0 0 360; loop: true; dir: alternate; dur: 10000;">
  <a-sphere position="0.5 1 1" radius="0.1" color="rgb(200,30,100)"></a-sphere>
</a-entity>
```



Modifique o código para criar a rotação da esfera em torno dos eixos x e y.

Para deixar o eixo de rotação da Lua em função do centro da Terra, criamos as entitys: a principal, que engloba as duas esferas, com o centro da Terra como posição e rotação de 30° em torno do eixo z; a primeira entidade filha tem a esfera da Terra, com a rotação invertida de 30° e a animação na rotação de 0° e 360° em torno do eixo y; e a segunda filha com a esfera da Lua com uma translação de unidade 3 ao longo do eixo x e a animação de 360° a 0°.



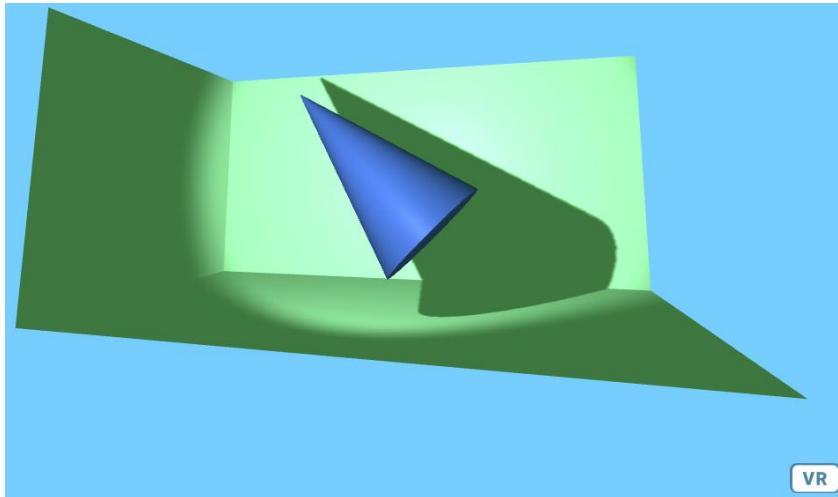
```
<a-entity position="0 2 -4" rotation="0 0 30">
  <a-entity rotation="0 0 -30">
    <a-sphere src="#textura1" scale="2 2 2" animation="property: rotation; from: 0 0 0;
      to: 0 360 0; loop: true; dur: 10000; easing: linear;"></a-sphere>
  </a-entity>
  <a-entity animation="property: rotation; from: 0 360 0; to: 0 0 0; loop: true; dur: 5000;
    easing: linear">
    <a-sphere position="3 0 0" src="#textura2" scale="0.5 0.5 0.5"></a-sphere>
  </a-entity>
</a-entity>
```

As animações nas cenas programadas em Realidade Virtual podem envolver outras propriedades, tais como **scale**, **position**, **color**, **width**, **height**, **depth**, **opacity** e **light**. Na tag mostrada a seguir, temos a animação com mudança de cor de um isosaedro.

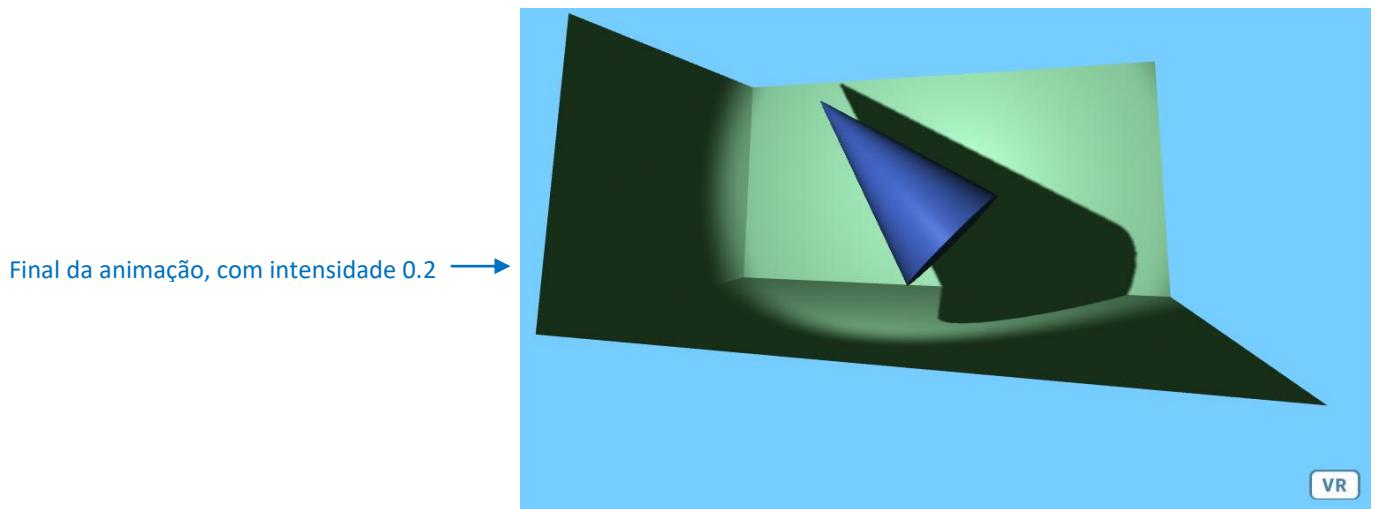
```
<a-icosahedron color="blue" position="1 1.5 -4" radius="1.5" animation="property:
  components.material.material.color; type: color; to: red; loop: true; dir: alternate;
  dur: 5000;"></a-icosahedron>
```

A animação da iluminação do tipo **hemisphere** pode ser feita usando a tag mostrada a seguir, animando **position** ou **intensity** com um cone e os planos que representamos nos exemplos anteriores.

```
<a-light type="hemisphere" color="#eaeaea" light="groundColor:green" intensity="0.7"
  animation="property: intensity; to: 0.2; loop: true; dir: alternate; dur: 5000;"></a-light>
```



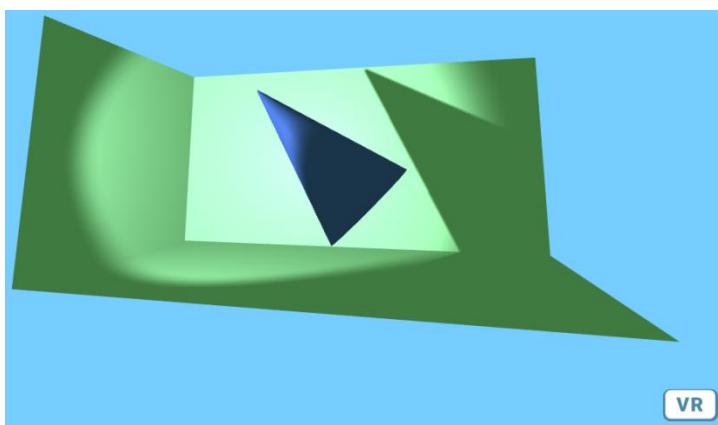
VR



VR

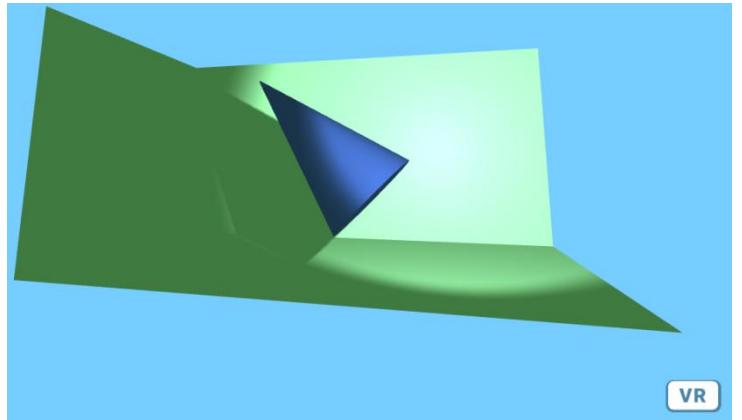
A animação da iluminação do tipo **spot** pode ser feita usando a tag mostrada a seguir, animando-se a propriedade **position** ou a propriedade **intensity**.

```
<a-light type="spot" intensity="0.75" angle="45" penumbra="0.2" light="castShadow:true"  
position="-2 2 -0.5" animation="property: position; to: 2 2 -0.5; loop: true; dir:  
alternate; dur: 10000;"></a-light>
```



VR

Fim da animação, com position (2, 2, -0.5) →



Experimente fazer animações com outros elementos na cena e outros tipos de iluminação também. Utilize a animação de iluminação no exemplo da Terra e da Lua.

8.3. Câmera

O componente da câmera define a partir de qual perspectiva o usuário vê a cena, além de possibilitar interações do usuário com objetos da cena. É comum que a câmera seja emparelhada com componentes de controle, os quais permitem que os dispositivos de entrada se movam e girem a câmera.

Podemos modificar a altura e a posição inicial do usuário na cena com a seguinte tag:

```
<a-camera position="-5 2 3"></a-camera>
```

Se a cena precisa de uma câmera fixa, sem a possibilidade de o usuário movimentar-se, basta usar a seguinte propriedade **look-controls-enabled**, que permite os movimentos do usuário na cena quando existir a conexão de um óculos de Realidade Virtual com o ambiente programado:

```
<a-camera position="-5 2 3" look-controls-enabled="false"></a-camera>
```

A propriedade **wasd-controls-enabled** permite que o usuário movimente-se na cena usando as teclas W (aproximação) A (direita) S (afastamento) e D (esquerda). Para bloquear esta funcionalidade, basta usar a seguinte tag:

```
<a-camera position="-5 2 3" wasd-controls-enabled="false"></a-camera>
```

Agora vamos estudar as formas de definir materiais no ambiente de Realidade Virtual. Nos exemplos mostrados anteriormente, definimos apenas a geometria dos elementos (raios, alturas, comprimentos, larguras) e cores. Aplicando a textura de um tronco de árvore em um cilindro, podemos definir a metalicidade e a rugosidade deste elemento com as seguintes tags:

```
<a-assets>
  
</a-assets>
```

```
<a-camera position="0 0 2"></a-camera>
<a-cylinder src="#arvore" position="0 2 0" radius="0.5" height="2" height="2" metalness="0"
  roughness="1" side="double"></a-cylinder>

<a-light type="ambient" color="white" intensity="0.4"></a-light>
<a-light type="directional" intensity="0.8" position="-1 0 0"></a-light>
```



Se inserirmos uma cor na tag do cilindro, a textura fica com a tonalidade da cor escolhida. Nos próximos exemplos, vamos utilizar um controle de órbita da biblioteca threeJS para a-frame. Este controle permite que os usuários girem a câmera em torno de um objeto ou de um ponto chamado de alvo (target). No cabeçalho do arquivo HTML, colocamos a referência da biblioteca:

```
<script src="https://unpkg.com/aframe-orbit-controls@1.3.0/dist/aframe-orbit-controls.min.js"></script>
```

Vamos inserir as propriedades de órbita na tag da câmera, com o alvo sendo aproximadamente o centróide dos elementos que vamos inserir na cena:

```
<a-camera orbit-controls="target: -1 1.5 1; minDistance: 0.5; maxDistance: 180; initialPosition: -1 1.6 3.5"></a-camera>
```

Vamos inserir a imagem equirectangular de fundo (sky) e transformá-la em uma imagem do tipo cubemap, ou seja, o mesmo formato que usamos nos ambientes do PyVista e VTK. Uma página que oferece a conversão de imagem panorâmica em cubemap é: <https://jaxry.github.io/panorama-to-cubemap/>

Neste projeto, vamos criar reflexos dos objetos construídos utilizando a propriedade reflection na tag a-scene com a iluminação criada na cena.

```
<a-scene reflection>
```

A iluminação da cena será feita com as seguintes tags:

```
<a-light type="ambient" color="#eaeaea" intensity="0.3"></a-light>
<a-light type="spot" intensity="0.75" angle="60" penumbra="0.5" shadow="cast: true; receive: false" position="-2 2 4"></a-light>
```

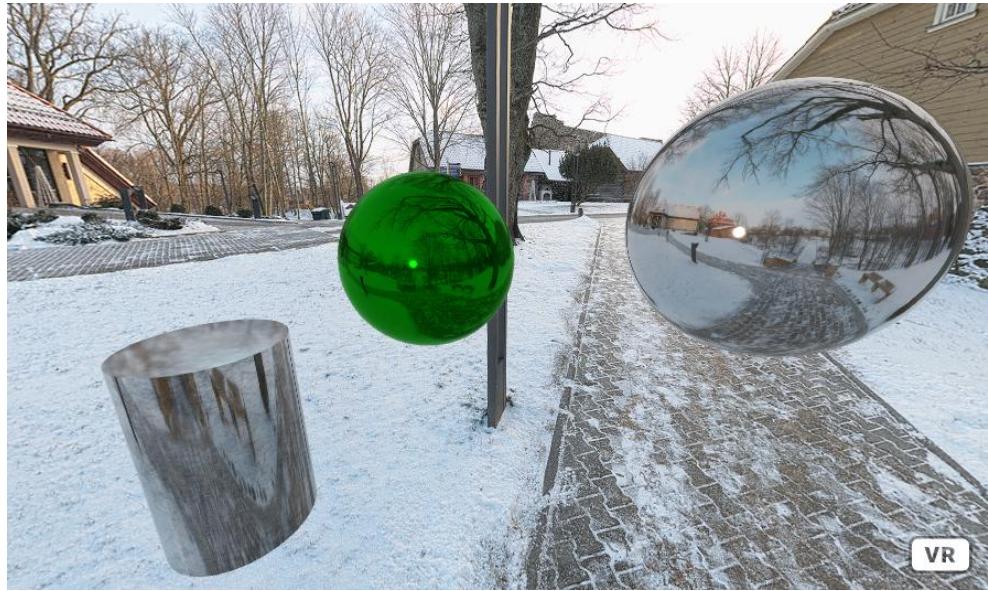
Dentro da tag assets, vamos inserir a imagem do céu, uma textura de metal e as imagens do céu em formato cubemap dentro de tag a-cubemap.

```
<a-assets>
  
  
  <a-cubemap id="ceu2">
    
    
    
    
    
    
  </a-cubemap>
</a-assets>
```

Agora podemos inserir as tags dos objetos com metalicidade 1 e rugosidade 0. A reflexão do ambiente fica restrita apenas à imagem do céu da cena. A propriedade envMap (mapa do ambiente) faz o reflexo do céu da cena em formato cubemap nos objetos da cena.

```
<a-sphere position="1 2 0.5" radius="1" side="double" color="silver" metalness="1" roughness="0" segments-height="36" shadow="" segments-width="64" material="envMap: #ceu2;"></a-sphere>
<a-sphere position="-2 1.5 -0.5" color="green" radius="1" side="double" metalness="1" roughness="0" segments-height="36" shadow="" segments-width="64" material="envMap: #ceu2;"></a-sphere>
<a-cylinder src="#metal" position="-3 0.5 1.5" color="white" radius="0.5" height="1.5" side="double" metalness="1" roughness="0" shadow="" material="envMap: #ceu2;"></a-cylinder>
```

Experimente modificar os materiais dos objetos da cena para observar as transformações neste projeto com reflexões.

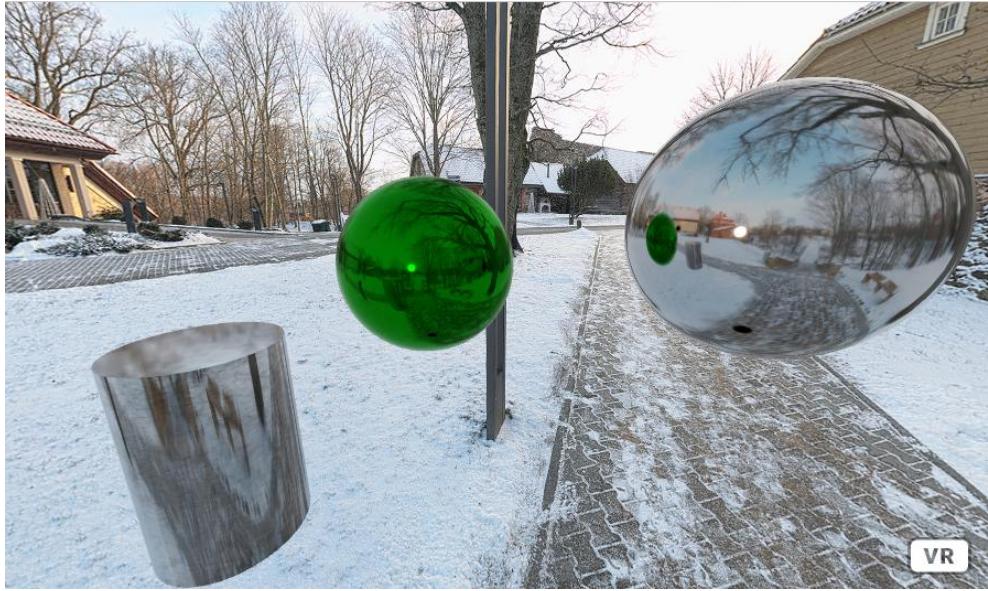


Usando a versão 1.0.4 do a-frame com a biblioteca cameracube do three.js, podemos criar as reflexões entre os elementos da cena. Vale lembrar que estes efeitos deixam a renderização um pouco mais lenta.

```
<script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
<script src="./java/camera-cube-env.js"></script>
```

Dentro das tags dos elementos que terão as reflexões, colocamos as seguintes propriedades:

```
camera-cube-env="distance: 500; resolution: 512; repeat: true; interval: 1;"
```



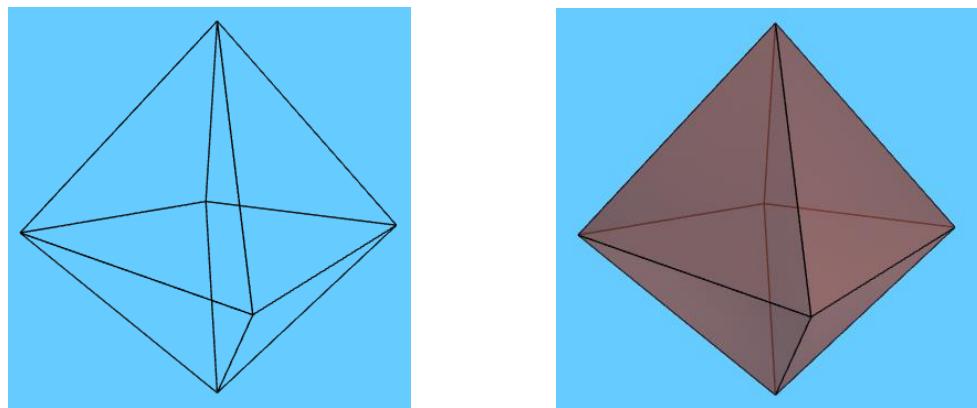
Experimente modificar os materiais dos objetos da cena para observar as transformações neste projeto de reflexões.

8.4. Poliedros

O atributo `wireframe` pode ser usado para deixar os objetos “aramados”, ou seja, somente as arestas aparecem na renderização. Este recurso fica interessante para estudar a geometria dos objetos representados no ambiente.

No primeiro exemplo mostrado a seguir, temos a representação do objeto com o `wireframe`. No segundo exemplo, o objeto está representado com uma tag de `wireframe` e a outra com as propriedades das faces.

```
<a-octahedron position="0 2 0" radius="2" color="black" wireframe="true"></a-octahedron>
<a-octahedron position="0 2 0" radius="2" side="double" color="tomato" metalness="0.6"
roughness="0.5" opacity="0.5"></a-octahedron>
```

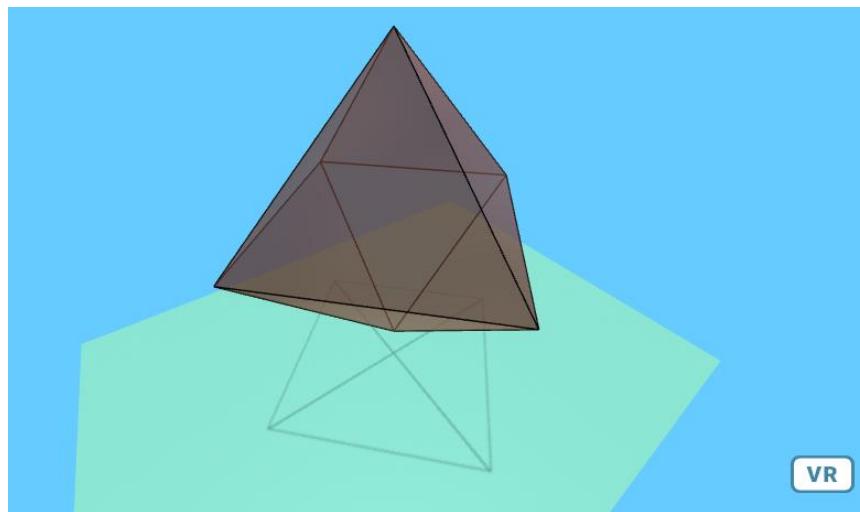


Com este material aramado, podemos usar a iluminação direcional e projetar as arestas dos sólidos em planos colocando as propriedades de sombras mostradas anteriormente. No exemplo a seguir, construímos um plano horizontal e colocamos a propriedade na tag do octaedro aramado para produzir as sombras das arestas. A luz direcional fica com target em Y para projetar ortogonalmente no plano horizontal.

```
<a-light type="directional" intensity="0.3" position="0 5 0" light="castShadow:true"
target="#directionaltargetY">
<a-entity id="directionaltargetY" position="0 -1 0"></a-entity>
</a-light>
```

Dentro da tag do octaedro aramado, colocamos a propriedade de sombra:

```
shadow="cast: true"
```



Podemos definir cores e padrões de materiais dos objetos em aframe dentro da tag assets. A tag que utilizamos para criar os conjuntos de propriedades comuns de objetos chama-se mixin, e funciona como propriedades do CSS (Cascading Style Sheets), que podem ser compartilhadas com vários objetos em uma cena. Lembre-se de criar um identificador para “linkar” os elementos com seus materiais.

No primeiro exemplo das reflexões, os três objetos compartilham algumas propriedades em comum, que podem ser agrupadas na seguinte tag:

```
<a-mixin id="padrao" material="metalness:1; roughness:0; side:double; envMap: #ceu2;" 
shadow=""></a-mixin>
```

Desta forma, podemos colocar as propriedades comuns dos três objetos com as tags:

```
<a-sphere position="1 2 0.5" radius="1" color="silver" segments-height="36" segments-width="64"
    mixin="padrao"></a-sphere>
<a-sphere position="-2 1.5 -0.5" color="green" radius="1" segments-height="36" segments-
    width="64" mixin="padrao"></a-sphere>
<a-cylinder src="#metal" position="-3 0.5 1.5" color="white" radius="0.5" height="1.5"
    mixin="padrao"></a-cylinder>
```

As tags ficam mais organizadas e fica mais fácil de encontrar as informações dos objetos de cada cena. Note que podemos criar várias tags `mixin`. Se criarmos uma tag com as propriedades comuns das geometrias das duas esferas da cena, obtemos:

```
<a-mixin id="padrao2" geometry="segments-height:36; segments-width:64; radius:1;"></a-mixin>
```

E as tags de esferas ficam assim:

```
<a-sphere position="1 2 0.5" color="silver" mixin="padrao padrao2"></a-sphere>
<a-sphere position="-2 1.5 -0.5" color="green" mixin="padrao padrao2"></a-sphere>
```

As construções de sólidos que não têm funções prontas no aframe podem ser feitas com a biblioteca faceset. A ideia é de construir as faces dos sólidos a partir de triangulações. Primeiro, precisamos referenciar a biblioteca no cabeçalho da página colocando a seguinte tag:

```
<script src="https://andreasblesch.github.io/aframe-faceset-component/dist/aframe-faceset-
    component.min.js"></script>
```

Vamos criar a base pentagonal de uma pirâmide com vértices de coordenadas A(0 0 0), B(1 0 0), C(1.31 0 0.95), D(0.5 0 1.5) e E(-0.3 0 0.95). O comando usado permite a definição do polígono por meio da sequência dos vértices que definem este polígono (como se estivéssemos desenhando o seu contorno). Para criar propriedades de cores, materiais e animações, vamos colocar os comandos da pirâmide em uma tag entity com `id`.

Os materiais do sólido aramado (com as arestas visíveis), das faces do sólido e dos rótulos dos vértices são definidas na tag assets:

```
<a-assets>
    <a-mixin id="aramado" material="color: black; wireframe: true; wireframe-linewidth:1;"></a-
        mixin>
    <a-mixin id="cor1" material="color: #d8ef09"></a-mixin>
    <a-mixin id="padrao" material="opacity: 0.5; side: double; metalness:0.3;
        roughness:0.9;"></a-mixin>
    <a-mixin id="texto" text="width: 4; side:double; color: black" rotation="-25 0 0">
</a-assets>
```

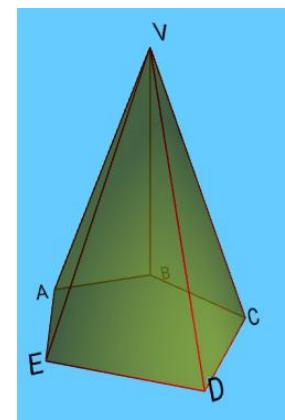
Se a pirâmide estiver com a base aberta, basta colocar as coordenadas do vértice principal V(0.5 6 0.7) após o vértice E na tag faceset.

Quando o novo vértice está em outro plano, precisamos colocar a direção das projeções ortogonais da face: é a definição da reta normal ao plano da face. Para colocar a direção de projeções, basta informar dentro da tag faceset a propriedade `projectdir`, com direção x, y ou z. No nosso caso, a direção de projeções é y.

Como a base está aberta, podemos usar o aramado (wireframe) para desenhar as arestas, repetindo a tag com `mixin aramado`. Se você quiser fechar a base, basta repetir a tag de vértices da base.

Os nomes dos vértices podem ser inseridos com a tag de texto. As coordenadas de cada tag devem ter valores próximos dos vértices, sempre ajustados para melhor visualização. Como eles começam sempre a contar da esquerda para a direita, nomes como do vértice E devem ter coordenada x um pouco menor do que a coordenada do respectivo vértice. Já o vértice C pode ter a coordenada x um pouco maior do que a coordenada de seu vértice.

```
<a-entity id="piramide_pentagonal_reta">
```



```

<a-text position="-0.2 0 0" value="A" mixin="texto"></a-text>
<a-text position="1.1 0 0" value="B" mixin="texto"></a-text>
<a-text position="1.31 0 0.95" value="C" mixin="texto"></a-text>
<a-text position="0.5 0 1.5" value="D" mixin="texto"></a-text>
<a-text position="-0.45 0 0.95" value="E" mixin="texto"></a-text>
<a-text position="0.5 2.1 0.7" value="V" mixin="texto"></a-text>
<a-entity faceset="vertices: 0 0 0 1 0 0 1.31 0 0.95 0.5 0 1.5 -0.31 0 0.95" mixin="cor1 padrao"></a-entity>
<a-entity faceset="vertices: 0 0 0 1 0 0 1.31 0 0.95 0.5 0 1.5 -0.31 0 0.95 0.5 2 0.7; projectdir:y" mixin="cor1 padrao"></a-entity>
<a-entity faceset="vertices: 0 0 0 1 0 0 1.31 0 0.95 0.5 0 1.5 -0.31 0 0.95 0.5 2 0.7; projectdir:y" mixin="aramado"></a-entity>
</a-entity>

```

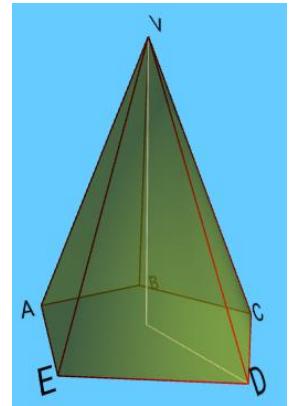
A ferramenta que podemos usar para desenhar segmentos na cena chama-se line. Cada linha deve ter seu respectivo nome, conforme mostra o exemplo a seguir: vamos desenhar a altura da pirâmide e o raio da base na cor branca. Precisamos apenas de 3 informações para desenhar a linha: o começo, o fim e a cor. As tags de linha devem ficar dentro da tag principal da pirâmide.

```

<a-entity line="start: 0.5 2 0.7; end: 0.5 0 0.7; color: white"
line_2="start: 0.5 0 0.7; end: 0.5 0 1.5; color: white"></a-entity>

```

Para desenhar prismas retos, podemos usar sempre a tag a-box, ajustando as medidas de largura, altura e profundidade. Porém, para prismas oblíquos a construção é similar às que vimos para pirâmides com a ferramenta faceset.



Objetos com geometria mais complexa podem ser construídos em outros softwares e importados para a cena do a-frame. Quando os objetos criados possuem muitas faces não convexas, precisamos construí-las em partes, todas com triangulações.

8.5. Interações com os objetos

Existem muitas formas de criar interação com os usuários em Realidade Virtual, pois existem diferentes plataformas e controles disponíveis. Trata-se de uma área em expansão, com muitos recursos sendo explorados.

Algumas formas simples de interações podem ser feitas com o cursor passando pelos elementos, ou criando efeitos gerados pelo click do mouse em determinadas regiões da cena. Estas formas são restritas à WEB 2D, onde podemos clicar em botões e arrastar objetos. Com a Realidade Virtual, podemos ampliar estas interações, pois podemos pegar, largar, jogar, esticar, além de outras interações.

A primeira maneira de interação é a de Orbitar objetos, que já utilizamos em alguns exemplos. Esta forma é indicada para interações em cenas com poucos objetos, onde o usuário pode usar controles dos óculos de Realidade Virtual para interagir, além do mouse e touch de smartphones e tablets. Desta forma, o usuário poderá interagir com o objeto que podemos utilizar também em cenas programadas com a tecnologia de Realidade Aumentada, com visualização de manipulação em Realidade Virtual.

Os controladores com 3 graus de liberdade (3DoF – 3 degrees of freedom) são limitados ao rastreamento rotacional. Os controladores 3DoF não têm rastreamento posicional, o que significa que não podemos estender a mão nem mover a mão para frente e para trás ou para cima e para baixo.

Os componentes do controlador 3DoF fornecem rastreamento rotacional, um modelo padrão correspondente ao hardware da vida real e eventos para abstrair os mapeamentos de botões. Os controladores para Google Daydream, Samsung GearVR e Oculus Go têm 3DoF e todos suportam apenas um controlador para uma mão. As tags para deixar a cena compatível com o uso destes controladores são:

```

<a-entity daydream-controls></a-entity>
<a-entity gearvr-controls></a-entity>
<a-entity oculus-go-controls></a-entity>

```

Controladores com 6 graus de liberdade (6DoF) possuem rastreamento rotacional e posicional. Ao contrário dos controladores com 3DoF que são restritos à orientação, os controladores com 6DoF podem se mover livremente no espaço 3D. Estes controles nos permitem chegar à frente, atrás, mover as mãos pelo corpo ou perto do rosto. Os controles 6DoF também se aplicam ao fone de ouvido e rastreadores adicionais (por exemplo, pés, adereços). Trata-se do recurso mínimo para fornecer uma experiência de VR totalmente imersiva.

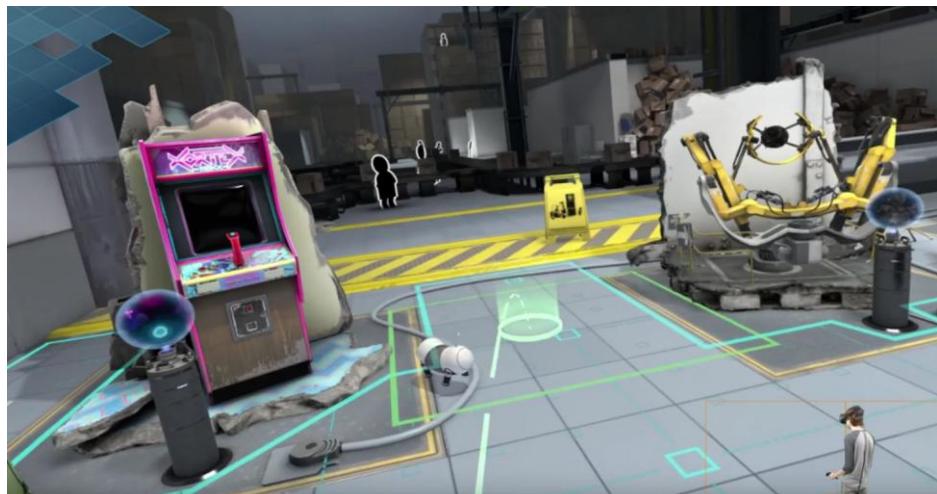
Os componentes do controlador 6DoF fornecem rastreamento completo, um modelo padrão correspondente ao hardware da vida real e eventos para abstrair os mapeamentos de botões. HTC Vive e Oculus Rift com Touch fornecem 6DoF e controladores para ambas as mãos. O HTC Vive também fornece rastreadores para rastrear objetos adicionais no mundo real em VR. As tags para deixar a cena compatível com o uso destes controladores são:

```
<a-entity vive-controls="hand: left"></a-entity>
<a-entity vive-controls="hand: right"></a-entity>
<a-entity oculus-touch-controls="hand: left"></a-entity>
<a-entity oculus-touch-controls="hand: right"></a-entity>
```

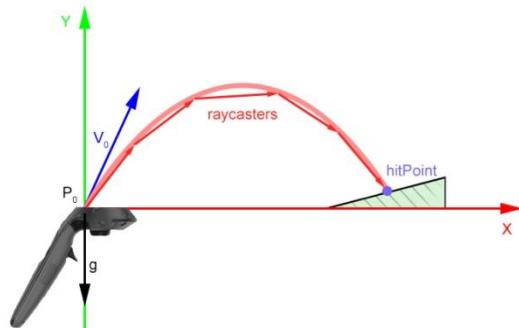
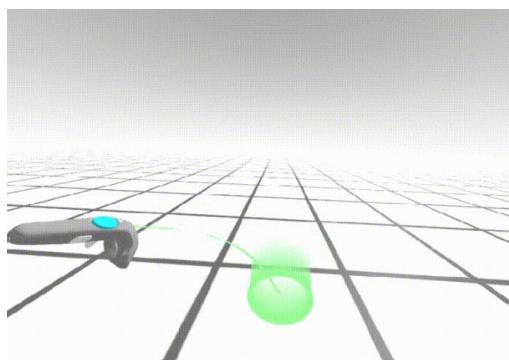
A ferramenta de interação de laser também pode ser inserida na cena, com possibilidade de escolha da cor do raio e distância do alcance:

```
<a-entity laser-controls raycaster="lineColor: red; lineOpacity: 0.5; far:5"></a-entity>
```

Outro recurso interessante da Realidade Virtual é o **teleporte**. Com o uso deste recurso, podemos evitar andar longas distâncias na cena, colocando a região de exploração da Realidade Virtual com maior alcance para o usuário. A ideia é simples: você mira em uma região e cria um alvo com o controle dos óculos, transportando seu referencial para este alvo.



A trajetória do laser para definir o alvo deve ser feita, de preferência, como um arco parabólico. Assim, alvos que ficam acima do usuário podem ser alcançados.



Com o recurso de teleporte, podemos escolher objetos da cena que podem ser bases de teleporte. Se os objetos não forem escolhidos, o plano $y = 0$ (piso) é usado como base. Dentro deste recurso, já existem validações

do ângulo da reta normal ao plano alvo com objetivo de evitar, por exemplo, que o usuário seja teleportado para uma parede. A tag de referência para o JavaScript é a seguinte:

```
<script src="https://fernandojsg.github.io/aframe-teleport-controls/dist/aframe-teleport-controls.min.js"></script>
```

A referência para câmeras dos óculos Vive e Rift como elemento do teleporte pode ser feita com as tags mostradas a seguir:

```
<a-entity id="cameraRig">
  <a-entity id="head" camera wasd-controls look-controls></a-entity>
  <a-entity id="left-hand" teleport-controls="cameraRig: #cameraRig; teleportOrigin:
    #head;"></a-entity>
  <a-entity id="right-hand" teleport-controls="cameraRig: #cameraRig; teleportOrigin:
    #head;"></a-entity>
</a-entity>
```

Na função `teleport-controls` podemos definir os tipos de objetos que permitem o teleporte. No exemplo mostrado a seguir, definimos com `id` box, escadas e piso nos assets, e a função fica assim:

```
teleport-controls="cameraRig: #cameraRig; teleportOrigin: #head; collisionEntities:
  [mixin='box'], [mixin='piso'], [mixin='escadas']"
```

As definições das câmeras para usar teleporte com óculos GearVR e Daydream são mostradas a seguir:

```
<a-entity id="cameraRig">
  <a-camera look-controls wasd-controls position="0 1 2"></a-camera>
  <a-entity teleport-controls="cameraRig: #cameraRig;" gearvr-controls></a-entity>
</a-entity>

<a-entity id="cameraRig">
  <a-camera look-controls wasd-controls position="0 1 2"></a-camera>
  <a-entity teleport-controls="cameraRig: #cameraRig;" daydream-controls></a-entity>
</a-entity>
```

Para testar o uso do teleporte, podemos criar uma cena simples, com 6 cubos e um plano para ser o piso. Podemos tentar movimentos de teleporte sobre o piso, que pode ser definido como um box com dimensões de largura e profundidade maiores, e altura bem reduzida. A função `teleport` não funciona para as tags `a-plane`, por isso podemos usar um box como piso na cena.

Nas tags do cabeçalho, inserimos as referências de bibliotecas de simulações de propriedades físicas, teleporte e interação:

```
<script src="https://unpkg.com/aframe-event-set-component@^4.1.1/dist/aframe-event-set-component.min.js"></script>
<script src="https://unpkg.com/super-hands@^3.0.3/dist/super-hands.min.js"></script>
<script src="https://fernandojsg.github.io/aframe-teleport-controls/dist/aframe-teleport-controls.min.js"></script>
<script src="https://rawgit.com/donmccurdy/aframe-physics-system/v3.2.0/dist/aframe-physics-system.min.js"></script>
<script src="https://unpkg.com/aframe-physics-extras@0.1.2/dist/aframe-physics-extras.min.js"></script>
```

Nas tags `a-scene`, colocamos a referência de propriedades físicas da cena. Para deixar os objetos movendo-se como se estivessem flutuando, basta usar a gravidade zero da seguinte forma: `physics="gravity: 0"`.

```
<a-scene physics>
```

As configurações de teleporte e interações com os óculos de RV ficam junto com a tag de câmera, com as seguintes referências:

```

Propriedades para o cursor do mouse ←
<a-entity id="cameraRig">
  <a-camera look-controls wasd-controls position="0 1 2" capture-mouse raycaster="objects: .cube" cursor="rayOrigin:mouse" static-body="shape: sphere; sphereRadius: 0.001" super-hands="colliderEvent: raycaster-intersection; colliderEventProperty: els; colliderEndEvent: raycaster-intersection-cleared; colliderEndEventProperty: clearedEls">
    </a-camera>

  <a-entity teleport-controls="cameraRig: #cameraRig; collisionEntities: [ mixin='cubo' , [ mixin='piso' ] " gearvr-controls daydream-controls>
    </a-entity> → Teleporte GearRV e DayDream

  <a-entity laser-controls raycaster="showLine:true; far:3;" line="color:rgb(0,255,0); opacity:0.33">
    </a-entity> → Raio laser

  <a-entity oculus-touch-controls="hand: left"></a-entity>
  <a-entity oculus-touch-controls="hand: right"></a-entity> → Controles de manipulação
</a-entity>

```

Na tag assets, definimos as configurações do piso (estático) e dos cubos (dinâmicos):

```

<a-assets>
  <a-mixin id="cubo" geometry="primitive: box; width: 0.5; height: 0.5; depth: 0.5;" hoverable grabbable stretchable draggable droppable event-set__hoveron="_event: hover-start; material.opacity: 0.7; transparent: true" event-set__hoveroff="_event: hover-end; material.opacity: 1; transparent: false" dynamic-body shadow="">
  </a-mixin>
  <a-mixin id="piso" geometry="primitive: box; width: 10; height: 0.3; depth: 10;" shadow="" static-body>
  </a-mixin>
</a-assets>

```

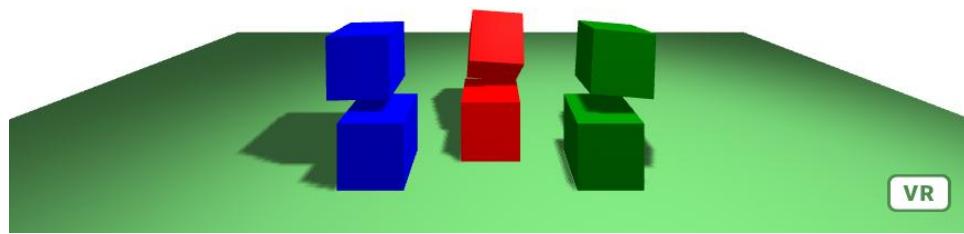
Fora da tag de assets, definimos o piso e os cubos:

```

<a-entity mixin="piso" class="piso" position="0 -1 0" material="color: #64B464"></a-entity>
<a-entity class="cubo" mixin="cubo" position="0 1 -1.25" material="color: red"></a-entity>
<a-entity class="cubo" mixin="cubo" position="0 1.6 -1.5" material="color: red"></a-entity>
<a-entity class="cubo" mixin="cubo" position="-0.9 1 -0.9" material="color: blue"></a-entity>
<a-entity class="cubo" mixin="cubo" position="-1 1.6 -1" material="color: blue"></a-entity>
<a-entity class="cubo" mixin="cubo" position="0.9 1 -0.9" material="color: green"></a-entity>
<a-entity class="cubo" mixin="cubo" position="1 1.6 -1" material="color: green"></a-entity>

```

Com a iluminação do tipo spot e ambient, temos a cena completa, com teleporte e interações com os objetos construídos. Modifique as configurações e tente movimentar os cubos com o cursor do mouse ou com um controle de óculos de RV.



Outras propriedades físicas permitem ajustar o amortecimento de colisões e a massa de cada objeto da cena. Podemos testar os valores de amortecimento linear e angular em cada um dos cubos para decidirmos qual valor ficará melhor na cena. Dentro da tag mixin dos cubos, removemos a propriedade dynamic-body para definir nas tags individuais dos cubos as propriedades de amortecimento.

No exemplo a seguir, o primeiro cubo vermelho tem as propriedades ajustadas:

```
<a-entity class="cubo" mixin="cubo" position="0 1 -1.25" material="color: red" dynamic-body="linearDamping:0.1; angularDamping:0.8; mass:0.5;"></a-entity>
```

Teste os valores possíveis dos amortecimentos entre 0 e 1. Se todos os objetos têm mesmas configurações físicas, podemos definir estas propriedades na tag `mixin`. A propriedade `sleepy` faz os movimentos dos objetos ficarem mais lentos, quase estáticos. Com a propriedade `sleepy` pode ser usada diretamente na tag de cada objeto da seguinte forma:

```
<a-entity class="cubo" mixin="cubo" position="0 1 -1.25" material="color: red" sleepy a-entity>
```

8.6. Importação de elementos

Agora vamos importar objetos 3D para as nossas cenas de RV. Temos dois tipos de arquivos que são aceitos pelo aframe. O mais comum é o GLTF (GL Transmission Format), que se trata de um projeto aberto da Khronos que fornece um formato comum, eficiente e extensível para objetos 3D. O componente `gltf-model` carrega um modelo 3D usando um arquivo com extensão `.gltf` ou `.glb`. Trata-se de uma especificação recente, que ainda está em expansão.

Em comparação com o formato OBJ, que suporta apenas vértices, normais, coordenadas de textura e materiais básicos, o GLTF fornece um conjunto mais poderoso de recursos. Além de todos os itens suportados pelo formato OBJ, o GLTF oferece:

- hierarquia de elementos;
- informações da cena, tais como fontes de luz e posições de câmeras;
- estrutura esquelética e de animação; e
- materiais e sombreamentos mais robustos.

Porém, quando temos modelos simples e sem animações, o formato OBJ continua sendo uma escolha comum e confiável. Podemos concentrar as definições de identificação e caminhos dos objetos GLTF ou OBJ na tag `assets`. Neste primeiro exemplo, temos um robô carregado com efeitos de sombra e textura metálica do próprio arquivo GLTF.

```
<a-assets>
    <a-asset-item id="objeto" src="objetos/robo/scene.gltf"></a-asset-item>
</a-assets>
```



Modelo criado por Artem Shupa-Dubrova: <https://sketchfab.com/fxtema>

A tag de inserção do objeto na cena pode ser colocada dentro de uma tag `entity`, que permite a inserção de mais objetos e fica mais fácil de manipular o conjunto de objetos dentro desta tag. Uma tag `a-box` com sombra foi colocada nesta cena junto com o robô.

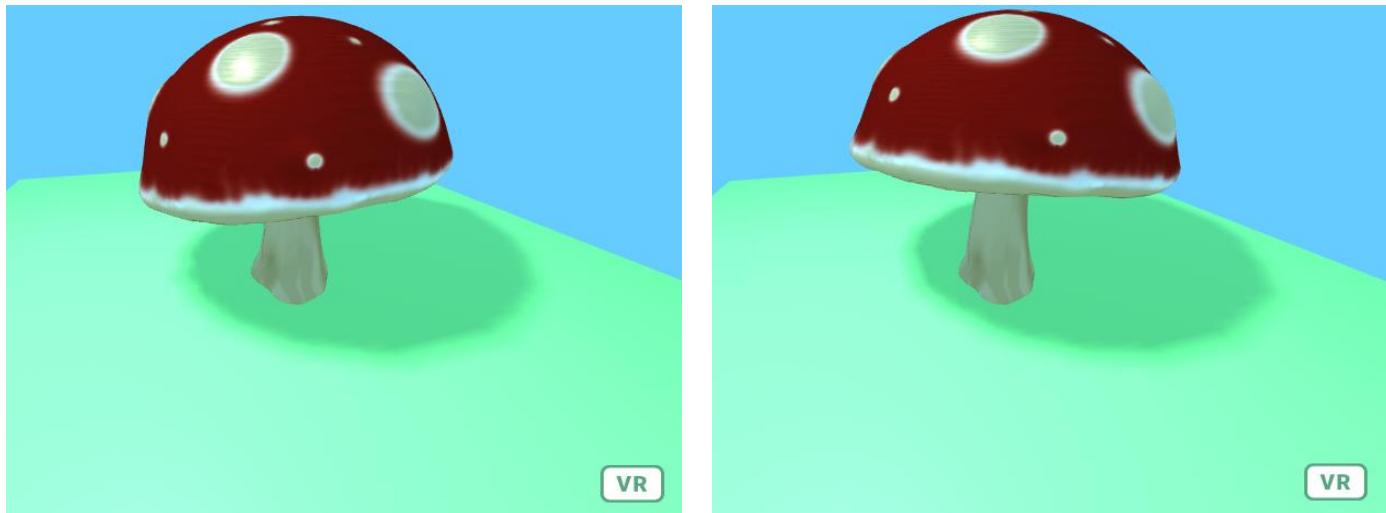
```
<a-entity position="0 0 0.5" scale="0.1 0.1 0.1">
  <a-gltf-model src="#objeto" shadow></a-gltf-model>
</a-entity>
<a-box scale="7 0.1 7" shadow color="rgb(100,180,100)"></a-box>
```

Quando o objeto tem a animação gerada no software de origem, utilizamos a tag de referência aframe-extras no cabeçalho da página:

```
<script src="https://cdn.jsdelivr.net/gh/donmccurdy/aframe-extras@v6.1.1/dist/aframe-extras.min.js"></script>
```

Além disso, colocamos a propriedade animation-mixer na tag do objeto GLTF:

```
<a-gltf-model src="#objeto" shadow animation-mixer></a-gltf-model>
```



Modelo criado por Aaron XR Dev: <https://sketchfab.com/AaronXRdev>



Modelo criado por AutoRunFail: <https://sketchfab.com/AutoRunFail>

As animações também podem ser feitas no aframe. Por exemplo, se criamos dois objetos GLTF ou OBJ separados, um deles com o corpo do avião e o outro com a hélice, a animação da hélice pode ser feita tomando-se o cuidado de usar as coordenadas de rotação e distância do objeto originais do software.

```
<a-assets>
  <a-asset-item id="objeto" src="objetos/aviao.glb"></a-asset-item>
```

```
<a-asset-item id="objeto1" src="objetos/helice.glb"></a-asset-item>
<a-mixin id="texto" text="color: black; align: left; width: 8; side:double;"></a-mixin>
</a-assets>
```



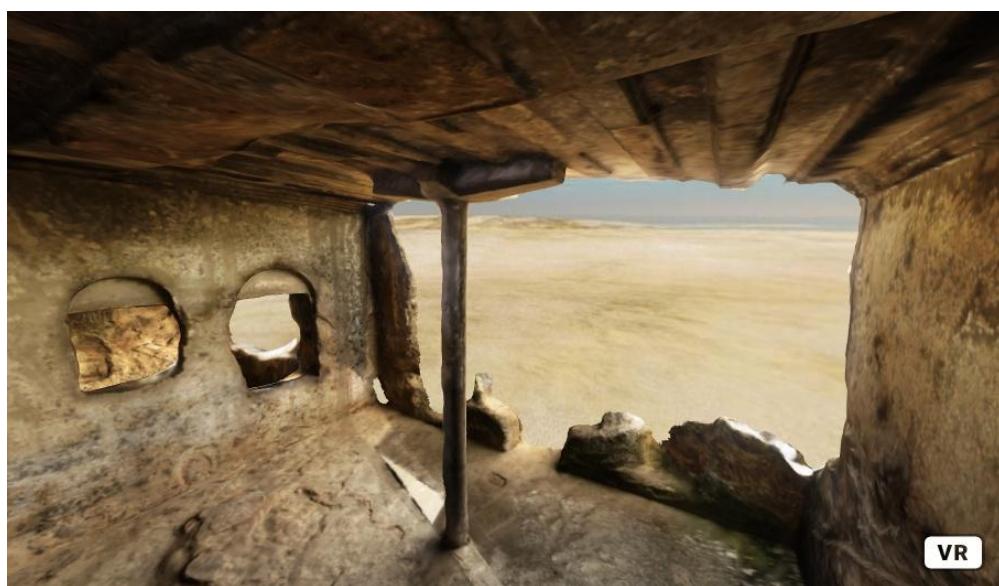
Modelo criado por Jeroen-Hut : <https://3dwarehouse.sketchup.com/user/0375467368344019534760919/Jeroen-Hut>

Usando a distância da hélice (objeto1) até a origem com a propriedade position, e a rotação em torno do eixo z com a propriedade rotation, criamos a animação dentro da tag a-gltf-model da hélice. Note que a tag-mãe controla a escala, posição e rotação do avião, da hélice e do texto.

```
<a-entity position="0 0 0" scale="0.8 0.8 0.8" rotation="0 -15 0">
  <a-gltf-model src="#objeto" shadow></a-gltf-model>
  <a-entity position="3.54 2.14 0.03" rotation="0 0 13.3">
    <a-gltf-model src="#objeto1" animation="property: rotation; to: 360 0 0; loop: true;
      dur: 5000; easing: linear" shadow></a-gltf-model>
  </a-entity>
  <a-text position="0 0.15 3.5" mixin="texto" value="A airplane | design: Jeroen Hut"
    shadow></a-text>
</a-entity>
<a-box scale="10 0.1 10" shadow position="0 -0.05 0" color="rgb(100,180,100)"></a-box>
```

As outras animações possíveis para objetos importados são de iluminação, posição e escala. As cores e propriedades de materiais só podem ser modificadas no software onde os objetos foram criados.

Quando inserimos modelos de ambientes, construções, paisagens ou monumentos, podemos inserir uma imagem de fundo na tag a-sky para experimentar a imersão na cena criada. Efeitos de iluminação e de textura do modelo criado ajudam na experiência imersiva do usuário. Para modelos de grandes extensões, a ferramenta de teleporte ajuda o usuário na exploração da cena.





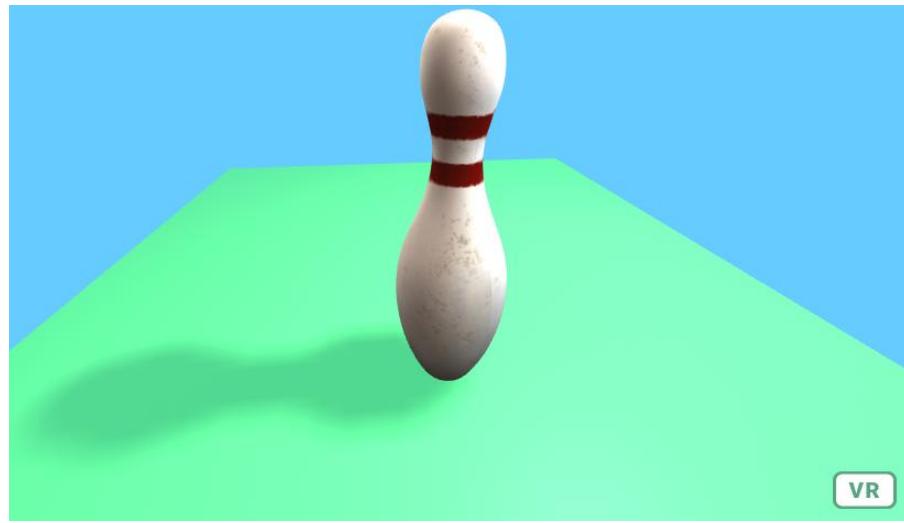
Modelo criado por Nik: <https://sketchfab.com/nikska>

A tag `obj-model` insere o modelo 3D criado com extensão `.obj` e seu respectivo arquivo de materiais `.mtl`. Já utilizamos os arquivos com extensão `OBJ` nas visualizações do VTK e PyVista, e colocamos as referências destes arquivos na tag `a-assets`:

```
<a-assets>
  <a-asset-item id="objeto" src="objetos/bowling.obj"></a-asset-item>
  <a-asset-item id="objeto-mtl" src="objetos/bowling.mtl"></a-asset-item>
</a-assets>
```

A tag de um pino de boliche em formato `OBJ` é a seguinte:

```
<a-entity position="0 0 0.5" scale="0.03 0.03 0.03" rotation="-90 0 0">
  <a-obj-model src="#objeto" mtl="#objeto-mtl" shadow></a-obj-model>
</a-entity>
```



Modelo criado por printable_models: https://free3d.com/user/printable_models

Vamos utilizar todos os elementos vistos nesta seção para criar uma cena com uma pista de boliche. Utilizaremos as referências de bibliotecas de teleporte, física e manipulação de objetos.

```
<head>
  <title>Pista de Boliche</title>
  <script src="https://aframe.io/releases/1.1.0/aframe.min.js"></script>
```

```

<script src="https://unpkg.com/aframe-event-set-component@^4.1.1/dist/aframe-event-set-component.min.js"></script>
<script src="https://unpkg.com/super-hands@^3.0.3/dist/super-hands.min.js"></script>
<script src="https://fernandojs.github.io/aframe-teleport-controls/dist/aframe-teleport-controls.min.js"></script>
<script src="https://rawgit.com/donmccurdy/aframe-physics-system/v3.2.0/dist/aframe-physics-system.min.js"></script>
<script src="https://unpkg.com/aframe-physics-extras@0.1.2/dist/aframe-physics-extras.min.js"></script>
</head>

```

Na tag `a-assets`, vamos definir os seguintes elementos, respectivamente:

- **bola** e **pino**: com as propriedades de manipulação e propriedade física dinâmica;
- **bow-gltf**: aquivo 3D de um pino de boliche;
- **plataforma** com a pista de rolagem: tag `a-box` de $18\text{m} \times 1,5\text{m}$ que permite o uso de teleporte, com referência de textura de madeira;
- **laterais** da plataforma: canaletas para a bola não sair da plataforma, textura de madeira;
- **piso** da cena: tag `a-box` de $30\text{m} \times 15\text{m}$ que permite o uso de teleporte, com textura de madeira;
- **céu** da cena: imagem equiretangular de uma casa de jogos de boliche;
- **piso1** e **piso2**: texturas de madeira do piso, da plataforma e das laterais da plataforma; e
- **point**: para usar com o raio laser e selecionar a bola para jogar na plataforma.

```

<a-assets>
  <a-mixin id="bola" geometry="primitive: sphere; radius: 0.3;" material="color: grey; metalness:0.8;" hoverable grabbable stretchable draggable droppable shadow dynamic-body="linearDamping:0.2; angularDamping:0.2; mass:7;" event-set_hoveron="_event: hover-start; material.opacity: 0.7; transparent: true" event-set_hoveroff="_event: hover-end; material.opacity: 1; transparent: false"></a-mixin>
  <a-mixin id="pino" scale="0.25 0.2 0.25" hoverable grabbable stretchable draggable droppable shadow dynamic-body="linearDamping:0.3; angularDamping:0.3; mass:0.5;"></a-mixin>
  <a-asset-item id="bow-gltf" src="objetos/bowling.gltf"></a-asset-item>
  <a-mixin static-body id="plataforma" geometry="primitive: box; height:0.1; width:18; depth:1.5;" material="src:#piso2; repeat:15 2; side:double; metalness:0.2; roughness:0.7" shadow></a-mixin>
  <a-mixin static-body id="lateral" geometry="primitive: box;" material="src:#piso2; repeat:15 1; side:double; metalness:0.2; roughness:0.7" shadow></a-mixin>
  <a-mixin static-body id="piso" geometry="primitive: box; height:0.1; width:30; depth:15;" material="src:#piso1; repeat:27 14; side:double; metalness:0.2; roughness:0.7" shadow></a-mixin>
  
  
  
  <a-mixin id="point" raycaster="showLine: false; objects: .pino, .bola" line="color:rgb(255,255,255); opacity:0.33;" static-body="shape: sphere; sphereRadius: 0.1" super-hands="colliderEvent: raycaster-intersection; colliderEventProperty: els; colliderEndEvent: raycaster-intersection-cleared; colliderEndEventProperty: clearedEls;"></a-mixin>
</a-assets>

```

As propriedades para manipular os objetos devem ser modificadas, incluindo as classes `bola` e `pino`. O teleporte pode conter somente o piso como elemento de colisão (também pode ser inserida a plataforma). Colocamos os controles dos óculos Rift, Gear e DayDream. O alvo da câmera é o mesmo mostrado em exemplos anteriores: `#cameraRig`.

```

<a-sky src="#ceu"></a-sky>

<a-entity id="cameraRig" position="3 1.6 2">

```

```

<a-camera id="head" look-controls capture-mouse cursor="rayOrigin:mouse" static-
  body="shape: sphere; sphereRadius: 0.1" super-hands="colliderEvent: raycaster-
  intersection; colliderEventProperty: els; colliderEndEvent: raycaster-intersection-
  cleared; colliderEndEventProperty: clearedEls;"></a-camera>
<a-entity teleport-controls="cameraRig: #cameraRig; collisionEntities: [ mixin='piso' ];
  teleportOrigin: #head; button: grip;" gearvr-controls daydream-controls></a-entity>
<a-entity laser-controls raycaster="showLine:true; far:3;" line="color:rgb(0,255,0);
  opacity:0.33;"></a-entity>
<a-entity oculus-touch-controls="hand: left"></a-entity>
<a-entity oculus-touch-controls="hand: right"></a-entity>
</a-entity>

```

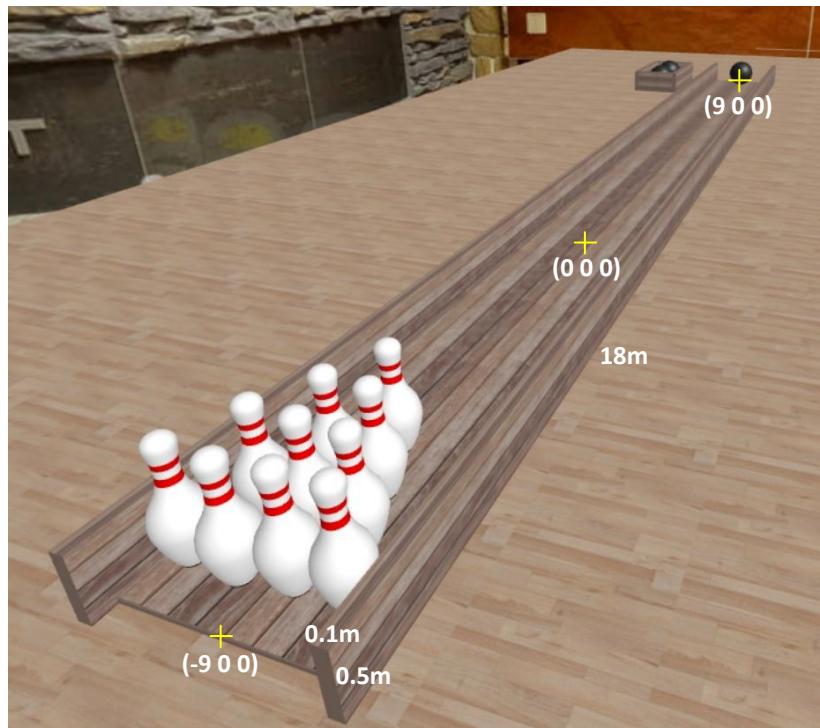
Usando a tag-mãe a-entity da pista, podemos movimentar todos os objetos mais facilmente com o atributo position. Neste caso, a melhor posição ficou com coordenada z = -2. Para não existir sobreposição de objetos, colocamos a plataforma com coordenada y = -0.05 e o piso com coordenada y = -0.1.

A criação das laterais da plataforma pode ser feita usando a ferramenta scale com valores 18 para x, 0.5 para y e 0.1 para z. Para ficar acima do piso, colocamos a coordenada y = 0.25. A barra da frente fica com coordenada z = 0.8 (se você quiser usar uma canaleta de 0.1m, pode colocar z = 0.9), pois a profundidade da plataforma é de 1.5. A barra de trás fica com z = -0.8.

```

<a-entity position="0 0 -2">
  <a-entity mixin="piso" position="0 -0.1 0"></a-entity>
  <a-entity mixin="plataforma" position="0 -0.05 0"></a-entity>
  <a-entity mixin="lateral" scale="18.3 0.5 0.1" position="0 0.2 -0.8"></a-entity>
  <a-entity mixin="lateral" scale="18.3 0.5 0.1" position="0 0.2 0.8"></a-entity>

```



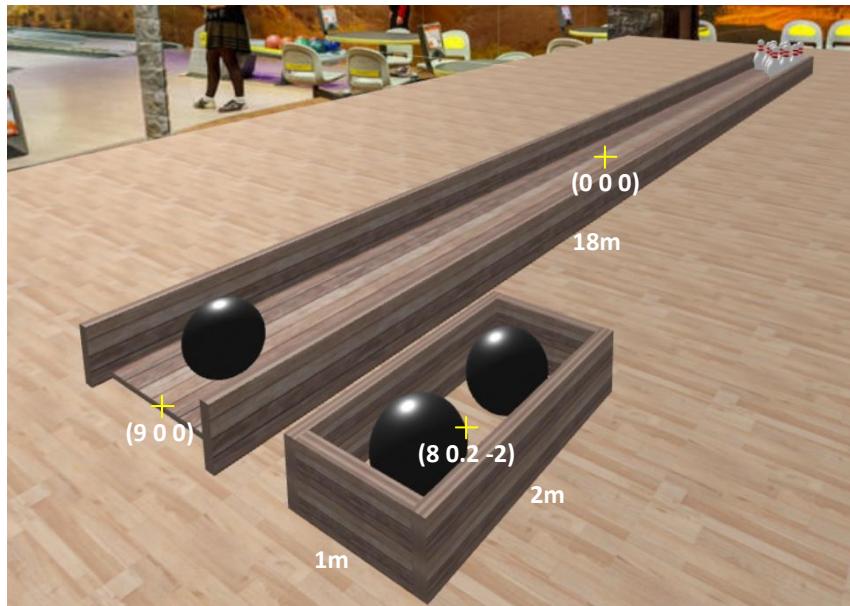
Como a pista tem 18m de comprimento e começa no ponto de coordenadas (0 0 0), os pinos terão distância menor do que 9m do centro da plataforma. O mesmo acontece com o suporte que colocaremos na outra extremidade da plataforma.

```

<a-entity id="suporte">
  <a-entity mixin="lateral" material="src:#piso2; repeat:2 2;" scale="2.1 0.5 0.1"
    position="8 0.2 -2.5"></a-entity>
  <a-entity mixin="lateral" material="src:#piso2; repeat:2 2;" scale="2.1 0.5 0.1"
    position="8 0.2 -1.5" rotation="0 0 0"></a-entity>
  <a-entity mixin="lateral" material="src:#piso2; repeat:2 2;" scale="0.9 0.5 0.1"
    position="7 0.2 -2" rotation="0 90 0"></a-entity>

```

```
<a-entity mixin="lateral" material="src:#piso2; repeat:2 2;" scale="0.9 0.5 0.1"
    position="9 0.2 -2" rotation="0 90 0"></a-entity>
```



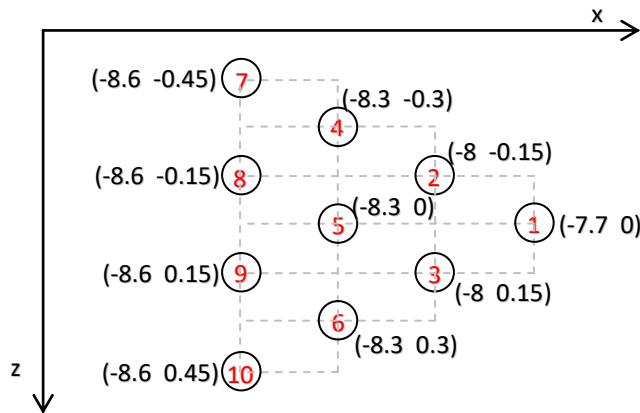
Como o suporte tem 2m de comprimento e 1m de profundidade, as coordenadas x serão iguais a 8. Para ficar atrás da plataforma, colocamos a coordenada z = -2 (centro do suporte). As coordenadas y serão iguais a 0.2.

As laterais do suporte têm coordenadas x iguais a 7 e 9. As coordenadas x são iguais a -2.5 e -1.5 para as partes da frente e de trás, e as laterais têm z = -2. As laterais precisam de rotações de 90° em torno do eixo y. As propriedades de escala podem ser de 2.1 para os comprimentos das partes da frente e de trás, e de 0.9 para as laterais, encaixando-se perfeitamente as 4 partes.

Podemos colocar 2 bolas no suporte e a terceira no começo da pista. As coordenadas podem ficar próximas dos valores que usamos no suporte e na plataforma:

```
<a-entity class="bola" mixin="bola" position="7.5 0 -2"></a-entity>
<a-entity class="bola" mixin="bola" position="8 0 -2"></a-entity>
<a-entity class="bola" mixin="bola" position="8.5 0 0"></a-entity>
```

As coordenadas dos pinos têm valores $x \leq 9$. Considerando valores próximos das posições reais dos pinos, as distâncias entre os centros das bases devem ser aproximadamente de 30cm, com raios iguais a 7cm. As coordenadas x ficam no intervalo entre -8.6 e -7.7: o pino 1 tem x = -7.7, os pinos 2 e 3 têm x = -8, os pinos 4, 5 e 6 têm x = -8.3, e os pinos 7, 8, 9 e 10 têm x = -8.6. A linha que passa pelos pinos 1 e 5 tem coordenada z = 0: logo, os pinos 2 e 8 têm z = -0.15, os pinos 3 e 9 têm z = 0.15, o pino 4 tem z = -0.3, o pino 6 tem z = 0.3, o pino 7 tem z = -0.45 e o pino 10 tem z = 0.45.



As tags dos pinos ficam com as seguintes propriedades:

```

<a-entity position="0 0.75 0.1">
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.6 0 0.15" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.6 0 -0.15" mixin="pino"></a-
    entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.6 0 0.45" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.6 0 -0.45" mixin="pino"></a-
    entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.3 0 0" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.3 0 -0.3" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8.3 0 0.3" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8 0 0.15" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-8 0 -0.15" mixin="pino"></a-entity>
  <a-entity class="pino" gltf-model="#bow-gltf" position="-7.7 0 0" mixin="pino"></a-entity>
</a-entity>

```

Agora podemos testar os movimentos no ambiente completo e inserir as tags de iluminação. Testando com o headset, smartphone, tablet ou no computador, temos a cena renderizada. Experimente modificar os elementos da cena (texturas, posição e iluminação) para melhorar o ambiente deste projeto.



Atividade 8

- 8.1. Utilize a ferramenta faceset ou line para construir uma pirâmide oblíqua de base hexagonal, com vértices A(2.5, 0, 0), B(3.5, 0, 0), C(4, 0, 0.87), D(3.5, 0, 1.73), E(2.5, 0, 1.73), F(2, 0, 0.87) e V(4.5, 2, 0.5). Insira elementos de sombra e um plano abaixo da base, para testar os efeitos de sombra e wireframe.
- 8.2. Utilize a ferramenta faceset ou line para construir um prisma oblíquo de base quadrada, com vértices de uma base A(5, 0, 0), B(6, 0, 0), C(6, 0, 1) e D(5, 0, 1). O vértice correspondente de A da outra base é E(5.5, 2, 0.5). Determine os outros vértices da segunda base, insira elementos de sombra e um plano abaixo da base ABCD, para testar os efeitos de sombra e wireframe.
- 8.3. Crie uma cena em VR com elementos importados (obj, gltf ou glb), sombreamentos em um plano (piso), animações e uma imagem equiretangular de fundo (céu).

9. Realidade Aumentada

A tecnologia de Realidade Aumentada (RA) utiliza um dispositivo com câmera para inserir objetos 2D e 3D junto com o ambiente da imagem da câmera. Desta forma, temos a criação de camadas virtuais de objetos 3D e texto sobre a imagem da câmera em tempo real (SIQUEIRA, 2019). A tecnologia de RA faz a junção das imagens dos objetos reais da câmera de um dispositivo com os objetos virtuais.

Os objetos e ambientes criados em Realidade Virtual (RV) podem ser visualizados usando o recurso de Realidade Aumentada (RA). A diferença é que os objetos serão visualizados com uma câmera de um dispositivo (celular, tablet ou computador), simulando que estão em uma determinada posição na cena produzida pela câmera. Na Realidade Virtual, todos os elementos são virtuais, o que não acontece na Realidade Aumentada, onde apenas alguns objetos virtuais misturam-se com objetos reais.



Fonte: <https://www.pngfly.com/png-1tbww0/>

Uma das grandes vantagens do uso do aframe é a quantidade reduzida de linhas de código para criar uma cena em Realidade Aumentada, pois usamos praticamente todos os recursos de Realidade Virtual com algumas tags de referências das bibliotecas de JavaScript. Outra vantagem é que por tratar-se de uma página HTML, não precisa de instalação de aplicativos para ser visualizada. O desenvolvimento é completamente independente de aplicativos e recursos e funciona em qualquer smartphone ou tablet.

Existem basicamente 3 formas de criar um projeto de Realidade Aumentada:

- Rastreamento de imagem ou marcador QR code: os objetos aparecem na cena quando a página reconhece uma imagem ou um marcador QR code;
- Rastreamento de faces: os objetos aparecem na cena quando a página reconhece uma face; e
- RA baseada em localização: os objetos aparecem na cena quando a câmera estiver em uma localização pré-estabelecida (latitude e longitude).

O primeiro projeto de Realidade Aumentada que vamos utilizar foi desenvolvido por Hiukim Yuen (<https://github.com/hiukim/mind-ar-js>). Este projeto utiliza as bibliotecas do A-frame e do JavaScript para RA com 5 objetos disponíveis para escolha pelo usuário.

```

<head>
  <meta name="viewport" content="width=device-width, user-scalable=no, minimum-scale=1.0,
  maximum-scale=1.0">
  <script src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-js@1.1.4/dist/mindar-
  face.prod.js"></script>
  <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
  <script src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-js@1.1.4/dist/mindar-face-
  aframe.prod.js"></script>
  <script>
    document.addEventListener("DOMContentLoaded", function() {
      const list = ["glasses1", "glasses2", "hat1", "hat2", "earring"];
      const visibles = [true, false, false, true, true];
      const setVisible = (button, entities, visible) => {
        if (visible) {
          button.classList.add("selected");
        } else {
          button.classList.remove("selected");
        }
        entities.forEach((entity) => {
          entity.setAttribute("visible", visible);
        });
      }
      list.forEach((item, index) => {
        const button = document.querySelector("#" + item);
        const entities = document.querySelectorAll("." + item + "-entity");
        setVisible(button, entities, visibles[index]);
        button.addEventListener('click', () => {
          visibles[index] = !visibles[index];
          setVisible(button, entities, visibles[index]);
        });
      });
    });
  
```

```

        });
    })
</script>
<style>
  body {margin: 0;}
  .example-container {overflow: hidden; position: absolute; width: 100%; height: 100%;}
  .options-panel {position: fixed; left: 0; top: 0; z-index: 2;}
  .options-panel img {border: solid 2px; width: 50px; height: 50px; object-fit: cover;
    cursor: pointer;}
  .options-panel img.selected {border-color: green;}
</style>
</head>

```

No cabeçalho de uma página com RA, temos que limitar a escala da tela, deixando-a fixa por meio dos atributos minimum-scale e maximum-scale, ambos iguais a 1. Temos que limitar a escala, pois alguns navegadores podem inserir os objetos da cena com escalas diferentes, afetando a visualização de RA. Além disso, colocamos as referências das bibliotecas do A-frame e o código JavaScript para inserir os objetos na cena.

```

<body>
  <div class="example-container">
    <div class="options-panel">
      
       ← Janela de opções dos objetos
      
      
      
    </div>
    <a-scene mindar-face embedded color-space="sRGB" renderer="colorManagement: true,
      physicallyCorrectLights" vr-mode-ui="enabled: false" device-orientation-permission-
      ui="enabled: false">
      <a-assets>
        <a-asset-item id="headModel" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/sparkar/head0occluder.gltf"></a-asset-item>
        <a-asset-item id="glassesModel" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/glasses/scene.gltf"></a-asset-item>
        <a-asset-item id="glassesModel2" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/glasses2/scene.gltf"></a-asset-item>
        <a-asset-item id="hatModel" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/hat/scene.gltf"></a-asset-item>
        <a-asset-item id="hatModel2" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/hat2/scene.gltf"></a-asset-item>
        <a-asset-item id="earringModel" src="https://cdn.jsdelivr.net/gh/hiukim/mind-ar-
          js@1.1.4/examples/face-tracking/assets/earring/scene.gltf"></a-asset-item>
      </a-assets>
      <a-camera active="false" position="0 0 0"></a-camera>
      <a-entity mindar-face-target="anchorIndex: 168"> ← Inserção dos objetos escolhidos
        <a-gltf-model mindar-face-occluder position="0 -0.3 0.15" rotation="0 0 0"
          scale="0.065 0.065 0.065" src="#headModel"></a-gltf-model>
      </a-entity>
      <a-entity mindar-face-target="anchorIndex: 10">
        <a-gltf-model rotation="0 -0 0" position="0 1.0 -0.5" scale="0.35 0.35 0.35"
          src="#hatModel" class="hat1-entity" visible="false"></a-gltf-model>
      </a-entity>
      <a-entity mindar-face-target="anchorIndex: 10">
        <a-gltf-model rotation="0 -0 0" position="0 -0.2 -0.5" scale="0.008 0.008 0.008"
          src="#hatModel2" class="hat2-entity" visible="false"></a-gltf-model>
      </a-entity>
      <a-entity mindar-face-target="anchorIndex: 168">
        <a-gltf-model rotation="0 -0 0" position="0 0 0" scale="0.01 0.01 0.01"
          src="#glassesModel" class="glasses1-entity" visible="false"></a-gltf-model>
      </a-entity>
      <a-entity mindar-face-target="anchorIndex: 168">

```

```

<a-gltf-model rotation="0 -90 0" position="0 -0.3 0" scale="0.6 0.6 0.6"
    src="#glassesModel2" class="glasses2-entity" visible="false"></a-gltf-model>
</a-entity>
<a-entity mindar-face-target="anchorIndex: 127">
    <a-gltf-model rotation="-0.1 -0 0" position="0 -0.3 -0.3" scale="0.05 0.05 0.05"
        src="#earringModel" class="earring-entity" visible="false"></a-gltf-model>
</a-entity>
<a-entity mindar-face-target="anchorIndex: 356">
    <a-gltf-model rotation="0.1 -0 0" position="0 -0.3 -0.3" scale="0.05 0.05 0.05"
        src="#earringModel" class="earring-entity" visible="false"></a-gltf-model>
</a-entity>
</a-scene>
</div>
</body>

```



A página HTML com Realidade Aumentada usa o recurso de webcam dos dispositivos e na primeira vez que a página for acessada, precisa da permissão do visitante. Por este motivo, a hospedagem da página com Realidade Aumentada deve ser feita em um servidor com protocolo de segurança **HTTPS**.

Um dos servidores HTTPS gratuito, que é usado para desenvolvimento de recursos computacionais de diversos tipos é o **github**. Podemos criar as pastas com os elementos de texturas, objetos e os próprios códigos HTML no servidor seguro github. O endereço fornecido por este servidor é simples e pode ser distribuído para os visitantes de seu site com Realidade Aumentada.

Em smartphones, as imagens dos objetos criados com RA podem ser visualizadas com velocidade de 60 frames por segundo e a resolução depende da qualidade de cada arquivo de textura usado. Falhas em texturas podem aparecer em alguns arquivos, dependendo também da velocidade de conexão. Recomenda-se que os arquivos sejam testados antes em RV para detectar falhas de texturas ou erros de programação.

Nosso segundo projeto de RA é de uma cena de Realidade Aumentada baseada na localização do dispositivo. A ideia é mostrar um objeto 2D ou 3D na posição das coordenadas de latitude e longitude indicadas. O objeto aparecerá sempre em uma mesma posição que você indica no código de programação.

No cabeçalho da página, inserimos as referências das bibliotecas. A tag principal que devemos colocar no cabeçalho é a que já usamos em RV do A-frame. Junto com esta tag, colocamos a tag desenvolvida por Jeromee Etienne que transforma a cena de RV em RA.

```

<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <meta name="viewport" content="width=device-width, user-scalable=no, minimum-scale=1.0,
        maximum-scale=1.0">
    <script src="https://aframe.io/releases/1.2.0/aframe.min.js"></script>
    <script src="https://unpkg.com/aframe-look-at-component@0.8.0/dist/aframe-look-at-
        component.min.js"></script>

```

```

<script src="https://jeromeetienne.github.io/AR.js/aframe/build/aframe-ar.js"></script>
<script>
    THREEEx.ArToolkitContext.baseURL =
        'https://raw.githack.com/jeromeetienne/ar.js/master/three.js/'
</script>
</head>

```

Referências das bibliotecas

Depois inserimos as referências dos objetos que aparecerão no ambiente de RA na tag a-assets; logo depois, basta inserir os objetos com animação nativa do arquivo ou criada no a-frame.

```

<body style="margin: 0px; overflow: hidden;">
    <a-scene vr-mode-ui="enabled: false" embedded arjs='sourceType: webcam; sourceWidth:1280;
        sourceHeight:960; renderer='logarithmicDepthBuffer: true;' displayWidth: 1280;
        displayHeight: 960; debugUIEnabled: false;'>
        <a-assets>
            <a-asset-item id="modelo" src="objetos/helicoptero.glb"></a-asset-item>
            <a-asset-item id="modelo1" src="objetos/helicea.glb"></a-asset-item>
            <a-asset-item id="modelo2" src="objetos/heliceb.glb"></a-asset-item>
        </a-assets>
        <a-entity look-at="[gps-camera]" gps-entity-place="latitude: <INSIRA AQUI>; longitude:
            <INSIRA AQUI>;">
            <a-entity rotation="0 120 0" scale="0.3 0.3 0.3">
                <a-gltf-model src="#modelo"></a-gltf-model>
                <a-entity position="0 0 0">
                    <a-gltf-model src="#modelo1" animation="property: rotation; to: 0 360
                        0; loop: true; dur: 4000; easing: linear"></a-gltf-model>
                </a-entity>
                <a-entity position="0.037 1.947 5.267" rotation="10 0 0">
                    <a-gltf-model src="#modelo2" animation="property: rotation; to: 360 0
                        0; loop: true; dur: 3200; easing: linear"></a-gltf-model>
                </a-entity>
            </a-entity>
        </a-entity>
        <a-camera gps-camera rotation-reader></a-camera>
        <a-light type="ambient" color="white" intensity="1.5"></a-light>
        <a-light type="directional" color="white" intensity="2" position="-1 1 1"></a-light>
    </a-scene>
</body>

```

Caminhos dos objetos da cena

Propriedades dos objetos na cena

Câmera e iluminação



Modelo criado por Francisco Fernandes Silva Neto: <https://3dwarehouse.sketchup.com/user/u9305242a-0aaf-45c8-a445-3627bb17a616/Francisco-Fernandes-Silva-Neto>

Esta forma de construir a cena com localização pode ser usada em construções, museus ou pontos turísticos. A forma mais comum de utilizar a Realidade Aumentada é usando o reconhecimento de imagens ou marcadores do tipo QR Code. O A-frame possui mais de 200 marcadores criados no projeto ARTookKit5:

<https://github.com/artoolkit/ARToolKit5/tree/master/doc/patterns>

No cabeçalho, inserimos praticamente as mesmas tags do projeto anterior, com exceção das tags de localização. A tag de animação de arquivos GLTF também foi inserida neste projeto.

```
<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <meta name="viewport" content="width=device-width, user-scalable=no, minimum-scale=1.0,
    maximum-scale=1.0">
  <script src="https://aframe.io/releases/1.2.0/aframe.min.js"></script>
  <script src="https://jeromeetienne.github.io/AR.js/aframe/build/aframe-ar.js"></script>
  <script src="https://cdn.jsdelivr.net/gh/donmccurdy/aframe-extras@v6.1.1/dist/aframe-
    extras.min.js"></script>
</head>
```

Referências das bibliotecas

Depois inserimos as referências dos objetos que aparecerão no ambiente de RA na tag a-assets; logo depois, basta inserir os objetos com animação nativa do arquivo ou criada no a-frame. Cada objeto deve estar na tag de marcador que você possui impresso, para que eles apareçam quando o dispositivo apontar para cada QR code programado na cena. Nesta cena, usaremos os marcadores Hiro para o helicóptero com animação do A-frame, #10 para um poste e #20 para a cena de animação nativa do arquivo GLTF.



```
<body style="margin: 0px; overflow: hidden;">
  <a-scene embedded arjs='sourceType: webcam; trackingMethod: best; detectionMode:
    mono_and_matrix; renderer='logarithmicDepthBuffer: true;' matrixCodeType: 3x3;
    debugUIEnabled: false;'>
    <a-assets>
      <a-asset-item id="modelo" src="objetos/helicoptero.gltf"></a-asset-item>
      <a-asset-item id="modelo1" src="objetos/helicea.gltf"></a-asset-item>
      <a-asset-item id="modelo2" src="objetos/heliceb.gltf"></a-asset-item>
      <a-asset-item id="modelo3" src="objetos/lamp/scene.gltf"></a-asset-item>
      <a-asset-item id="modelo4" src="objetos/evening/scene.gltf"></a-asset-item>
    </a-assets>
    <a-marker preset="hiro">
      <a-entity scale="0.5 0.5 0.5" position="0 0.5 0">
        <a-gltf-model src="#modelo"></a-gltf-model>
        <a-entity position="0 0 0">
          <a-gltf-model src="#modelo1" animation="property: rotation; to: 0 360 0;
            loop: true; dur: 4000; easing: linear"></a-gltf-model>
        </a-entity>
        <a-entity position="0.037 1.947 5.267" rotation="10 0 0">
          <a-gltf-model src="#modelo2" animation="property: rotation; to: 360 0 0;
            loop: true; dur: 3200; easing: linear"></a-gltf-model>
        </a-entity>
      </a-entity>
    </a-marker>
    <a-marker type="barcode" value="20">
      <a-entity scale="0.5 0.5 0.5" position="0 0.05 0">
        <a-gltf-model src="#modelo3"></a-gltf-model>
      </a-entity>
    </a-marker>
    <a-marker type="barcode" value="10">
      <a-entity scale="0.01 0.01 0.01" position="0 0.05 0">
    </a-entity>
  </a-scene>
```

Caminhos dos objetos da cena

Propriedades dos objetos do marcador Hiro

Propriedades do objeto do marcador #20

Propriedades do objeto do marcador #10

```

<a-gltf-model src="#modelo4" animation-mixer></a-gltf-model>
</a-entity>
</a-marker>

<a-entity camera></a-entity>
<a-light type="ambient" color="white" intensity="2"></a-light>
<a-light type="directional" color="white" intensity="2.5" position="-1 1 1"></a-light>
</a-scene>
</body>

```

O objeto com animação do helicóptero aparece um pouco acima do marcador Hiro. As propriedades position, scale e rotation podem ser usadas para posicionar os elementos com outras configurações. Não se recomenda usar coordenada y negativa, pois o objeto apareceria abaixo do marcador, o que pode causar confusão na visualização.



Modelo criado por Francisco Fernandes Silva Neto: <https://3dwarehouse.sketchup.com/user/u9305242a-0aaf-45c8-a445-3627bb17a616/Francisco-Fernandes-Silva-Neto>

O objeto com animação da maquete aparece acima do marcador #10. Modificando a orientação da tela do smartphone, conseguimos visualizar a maquete com mais detalhes.



Modelo criado por AutoRunFail: <https://sketchfab.com/AutoRunFail>

O terceiro objeto programado neste projeto, o poste de luz, aparece acima do marcador #20. Note que podemos inserir vários objetos em uma mesma cena programada, tomando-se o cuidado de não deixar os objetos muito grandes e nem os marcadores muito próximos.



Modelo criado por katsiaryna_kruhlenia: <https://sketchfab.com/kashiarita>

Atividade 9

- 9.1. Crie uma página de RA baseada em reconhecimento de faces para inserir outros objetos GLTF ou GLB.
- 9.2. Crie uma página de RA baseada em localização para inserir um objeto no ambiente.
- 9.3. Crie uma página de RA com pelo menos 4 objetos 3D mostrados em seus respectivos marcadores.

Referências

- A-frame. **A web framework for building 3D/AR/VR experiences**. Disponível em: <<https://aframe.io/>>, 2022.
- Anscombe, F. J. **Graphs in Statistical Analysis**. American Statistician, vol. 27, n. 1, p. 17–21, 1973.
- Card, S. K., Mackinlay, J. D., Shneiderman, B. **Readings in Information Visualization Using Vision to Think**. San Francisco: Browse books, 1999.
- Eler, D. M. **Visualização de Informação**. Disponível em: <<https://daniloeler.github.io/teaching/VISUALIZACAO/>>, 2020.
- Horst, A. M., Hill, A. P., Gorman, K. B. **Palmerpenguins: Palmer Archipelago (Antarctica) penguin data**. Disponível em: <<https://allisonhorst.github.io/palmerpenguins/>>. doi: 10.5281/zenodo.3960218, 2020.
- Keim, D. A. **Information Visualization and Visual Data Mining**. IEEE Transactions on Visualization and Computer Graphics, vol. 8, n. 1, p. 1–8, 2002.
- Keller, P. R, Keller, M. M. **Visual Cues: Pratical Data Visualization**. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- Moro, C. et al. **The effectiveness of virtual and augmented reality in health sciences and medical anatomy**. Anatomical sciences education, v. 10, n. 6, p. 549–559, 2017.
- Siqueira, P. H. **Desenvolvimento de ambientes web em Realidade Aumentada e Realidade Virtual para estudos de superfícies topográficas**. Revista Brasileira de Expressão Gráfica, v. 7, n. 2, p. 21–44, 2019.
- Shneiderman, B. **The eyes have it: a task by data type taxonomy for information visualization**. In: Proceedings of the 1996, IEEE Symposium on Visual Languages, p. 336–343. Washington, DC: IEEE Computer Society, 1996.
- Telea, A. C. **Data visualization: principles and practice**. Boca Raton: CRC Press, 2015.
- Ward, M., Grinstein, G.G., Keim, D. **Interactive data visualization foundations, techniques, and applications**. Massachusetts: A K Peters, 2010.
- Williams, J. G., Sochats, K. M., Morse, E. **Visualization**. Annual Review of Information Science and Technology (ARIST), v. 30, p. 161–207, 1995.