DISEÑO E IMPLEMENTACIÓN DE R3+ UNA BIBLIOTECA PARA GRAFICAR VECTORES Y CAMPOS VECTORIALES UTILIZANDO EL MOTOR GRÁFICO DE UNITY

JULIÁN DAVID MEDINA MOSQUERA

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
INGENIERÍA FÍSICA
POPAYÁN 2019

DISEÑO E IMPLEMENTACIÓN DE R3+ UNA BIBLIOTECA PARA GRAFICAR VECTORES Y CAMPOS VECTORIALES UTILIZANDO EL MOTOR GRÁFICO DE UNITY

JULIÁN DAVID MEDINA MOSQUERA

Trabajo de grado presentado como requisito parcial para optar al título de Ingeniero Físico.

Director

Dr. CARLOS ALBERTO RINCÓN LÓPEZ

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
INGENIERÍA FÍSICA
POPAYÁN 2019

NOTA DE ACEPTACIÓN

_____

_____

_____

Director _____
Dr. Carlos Alberto Rincón López

Jurado _____
Dra. Luz Elena Bolívar

Jurado _____
MSc. Jorge Washington Coronel

Fecha y lugar de sustentación: Popayán, 8 de noviembre de 2019.

AGRADECIMIENTOS:

A mis familiares y amigos que siempre han creído en mí, incluso cuando ni yo mismo me tengo fe.

RESUMEN:

En este trabajo se explica cómo implementar R3+, una biblioteca para Unity que permite hacer gráficas tridimensionales visualmente agradables, versátiles y con un alto rango de personalización, que permite graficar diversos tipos de elementos matemáticos en $\mathbb{R}^3$ como vectores, proyecciones cartesianas, esféricas y cilíndricas de vectores, operaciones de vectores (suma, resta, producto cruz), ángulo entre vectores, planos referentes a un vector (paralelo y ortogonal), campos vectoriales, superficies equipotenciales y ejes de referencia XYZ. Esta biblioteca puede ser implementada en cualquier proyecto desarrollado en Unity que requiera hacer uso de gráficas de los elementos matemáticos anteriormente mencionados.

Se hace una comparación entre herramientas similares como MATLAB, Wolfram o MatPlotLib; en cuestión de personalización, versatilidad, rendimiento, facilidad de implementación y licencias.

Palabras clave: vectores, campos vectoriales, graficas tridimensionales, Unity

CONTENIDO

CAPITULO I

INTRODUCCIÓN:

La abstracción matemática es inherente al ser humano. Herramientas como contar han existido incluso antes de la conceptualización abstracta de los números naturales. La representación gráfica de conceptos matemáticos es quizá el método más sencillo, acertado, práctico y ampliamente utilizado para llegar al entendimiento no abstracto de estos, partiendo de la gráfica como una herramienta de expresión directa no numérica. Elementos gráficos como los diagramas de tortas y barras permiten el entendimiento de magnitudes numéricas, sin la necesidad de un valor explícito; así mismo, la representación de funciones matemáticas en un plano cartesiano facilita la visualización de la evolución de una variable respecto a otra, sin tener que acudir a una tabla de valores.

Conforme la matemática evoluciona, también lo hacen los instrumentos que dan soporte a la comprensión de esta; desde el uso de papel milimetrado para hacer gráficas manualmente, pasando por los computadores analógicos que utilizaban un osciloscopio para mostrar gráficos en una pantalla, hasta las más modernas calculadoras graficadoras, la necesidad de expresar gráficamente elementos numéricos ha existido desde siempre. Con la llegada de la computación se logró automatizar el proceso de generar gráficas en dos dimensiones a partir de conjuntos de datos; sin embargo, cuando se busca hacer gráficas en tres dimensiones, existe una gran limitante en el alcance de las herramientas existentes.

En este trabajo se explica cómo implementar R3+, una biblioteca que utiliza el motor gráfico de Unity para generar diversos tipos de gráficas vectoriales en un espacio tridimensional, abriendo así un campo de posibilidades para el entendimiento no abstracto de la matemática vectorial en $\mathbb{R}^3$.

CAPITULO II

HERRAMIENTAS PARA GENERAR GRÁFICAS VECTORIALES:

Actualmente existen pocas herramientas que permitan hacer gráficas de elementos vectoriales en $\mathbb{R}^3$ que sean visualmente agradables, personalizables, de fácil implementación y computacionalmente eficientes y además puedan ser implementadas en diferentes proyectos.

Cabe mencionar que los graficadores de vectores y campos vectoriales no se pueden incluir dentro de la misma categoría que R3+, dado que tienen una función específica; no son herramientas de desarrollo que puedan ser implementadas en proyectos más grandes.

Si bien existen otras alternativas a la solución propuesta como *MATLAB*, *Wolfram*, o la biblioteca *MatPlotLib* para *Python*, estas poseen varias limitaciones en comparación con R3+. La principal limitación es el hecho de que estas plataformas no contienen motores gráficos especializados en la renderización de modelos 3D y los que implementan no están muy optimizados, por lo que pueden llegar a ser bastante pesados a la hora de renderizar muchos elementos al tiempo. Otra sobresaliente ventaja a las alternativas existentes es la compatibilidad con plataformas móviles, pues Unity permite exportar de forma nativa aplicaciones a los sistemas operativos móviles mas populares, Android y iOS; a diferencia a las herramientas existentes, que solo permiten trabajar en computador o web.

Analicemos las 3 herramientas de desarrollo más populares y la alternativa propuesta: R3+:

MATLAB: Es una de las herramientas más ampliamente utilizada en el mundo del análisis de datos. Este software permite generar una gran cantidad de gráficas con un buen grado de personalización; sin embargo, no implementa funciones directas para hacer gráficas de vectores, operaciones vectoriales, planos, o ángulos como si lo hace R3+. Si bien MATLAB incluye una función para hacer gráficas de campos vectoriales, esta es complicada de implementar, pues requiere varias líneas de código adicional para poder funcionar. En la Fig. 1 se aprecia un campo graficado utilizando MATLAB; este es poco personalizable, dado que los vectores son graficados como líneas en el espacio, sin volumen ni color variable; además, el motor gráfico que implementa MATLAB es bastante pesado, por lo que las gráficas resultantes, por simples que sean, son bastante pesadas, y requieren un hardware bastante poderoso para poder renderizar. MATLAB requiere una licencia para poder operar.

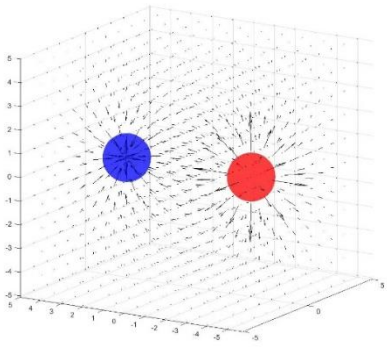


Fig. 1: Campo vectorial graficado con MATLAB

WOLFRAM: Es una herramienta ligera, que permite hacer gráficas en un computador, o desde la web sin necesidad de descargar ningún programa. Utilizando Wolfram Mathematica se pueden hacer muchos tipos de gráficas; sin embargo, este lenguaje no implementa funciones directas para hacer gráficas de vectores, operaciones vectoriales, planos, o ángulos como si lo hace R3+. Esta herramienta implementa una función directa y sencilla para hacer gráficas de campos vectoriales, sin embargo, estos son poco personalizables en términos de color y forma. En la Fig. 2 se aprecia un campo vectorial graficado con Mathematica. Se pueden colorear los vectores con algunos colormap, pero a diferencia de R3+, estos carecen de transparencia adaptativa, y corrimientos de color. Wolfram implementa un motor de renderizado bastante ligero en comparación a MATLAB, pero sigue siendo bastante limitado si se compara con el de Unity. Wolfram requiere una licencia para poder operar.

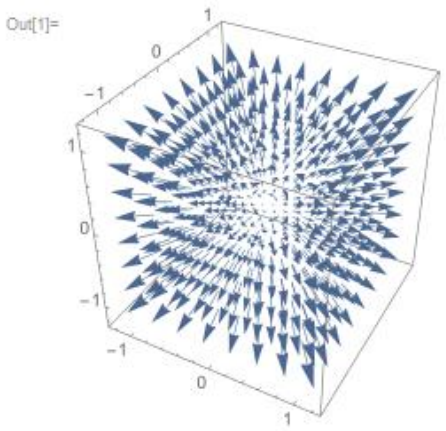


Fig. 2 Campo vectorial graficado con Mathematica

MatPlotLib: Es una biblioteca para Python inspirada en MATLAB. Es ligera, gratuita y sirve para generar una gran cantidad de gráficas. Esta biblioteca no implementa funciones directas para hacer gráficas de vectores, operaciones vectoriales, planos, o ángulos como si lo hace R3+. Esta herramienta contiene una función directa para hacer gráficas de campos vectoriales, pero al igual que MATLAB, requiere de varias líneas de código adicional para poder ser implementada. En la Fig. 3 se aprecia un campo vectorial graficado con MatPlotLib. A pesar de que el motor de renderizado de Python es bastante ligero, e incluso comparable al de Unity, los campos graficados son poco personalizables en términos de colormap y forma de los vectores.

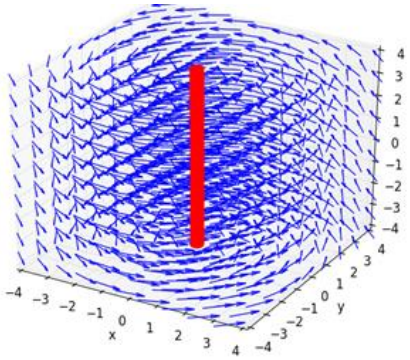


Fig. 3 Campo vectorial graficado con MatPlotLib

Unity: Es un motor gráfico especializado en el desarrollo de video juegos, por lo cual es bastante eficiente al momento de renderizar gráficos tridimensionales; sin embargo, no existe hasta ahora una biblioteca que utilice este motor para generar gráficos de elementos vectoriales; aquí es donde entra R3+ como la primer herramienta que permite explotar las capacidades de Unity para generar de una manera fácil y dinámica diversos tipos de gráficas vectoriales.

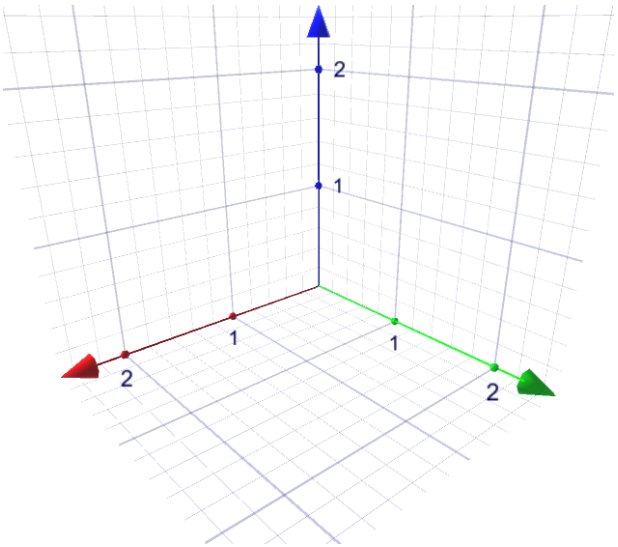
En el capítulo siguiente se expondrá como implementar cada uno de los elementos visuales de la biblioteca R3+, el cual es el objetivo general del trabajo.

CAPITULO III

IMPLEMENTACIÓN DE LA BIBLIOTECA PARA UNITY:

1. Graficar Ejes:

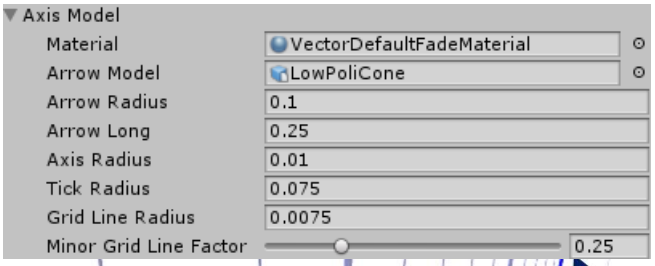
R3+ permite graficar ejes de una manera dinámica, sencilla y personalizable tanto por código como por inspector. Estos ejes se pueden personalizar en términos de color, forma, visualización, largo de ejes, disposición de ejes, distancia entre subdivisiones, visibilidad de grilla, y etiquetas.

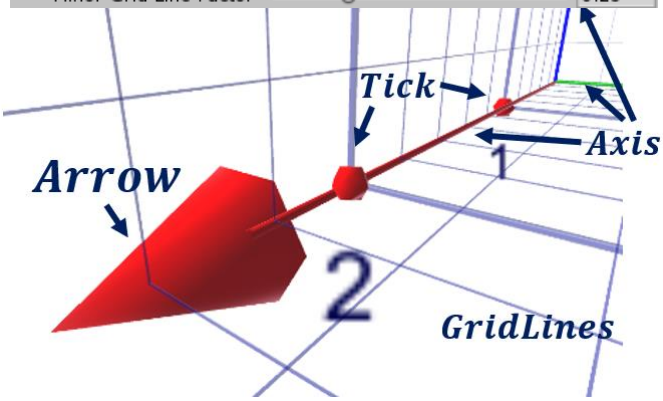


1.1. Agregar ejes a través de inspector:

I. Crear un nuevo *GameObject*
II. Agregar el componente "*Axes*"

1.2. Modificar ejes a través de inspector:





1.2.1. Axis Model: Indica la forma en que se va a renderizar el eje en la escena.

1.2.1.1. Material: Material que tendrán todos los modelos que conforman el eje.

1.2.1.2. ArrowModel: Modelo que se utiliza para renderizar las flechas de los ejes.

1.2.1.3. ArrowRadius: Radio de la flecha.

1.2.1.4. ArrowLong: Largo de la flecha.

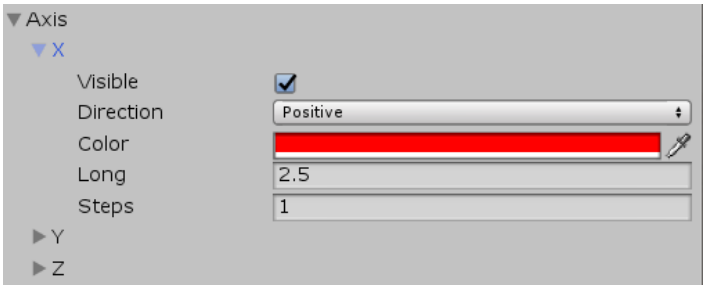
1.2.1.5. AxisRadius: Radio del cuerpo del eje.

1.2.1.6. TickRadius: Radio de los tick del eje.

1.2.1.7. GridLineRadius: Radio de las grillas principales.

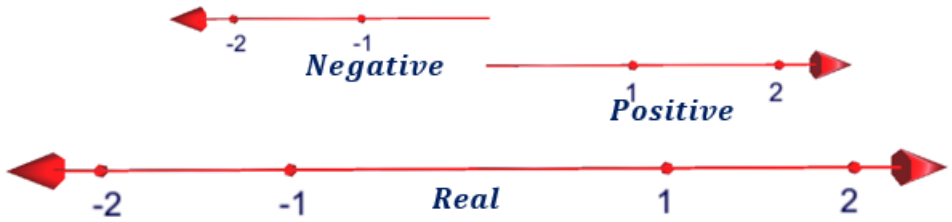
1.2.1.8. MinorGridLineFactor: Valor que varía entre 0 y 1, y define el grosor de las grillas menores; donde 0 es invisible y 1 es el mismo grosor de la grilla principal.

1.2.2. Axis: Define como se renderizan las líneas de cada uno de los ejes. Cada eje *X, Y, Z* se puede manejar por separado, y es personalizable en términos de visibilidad, dirección, color, largo y distancia entre ticks.



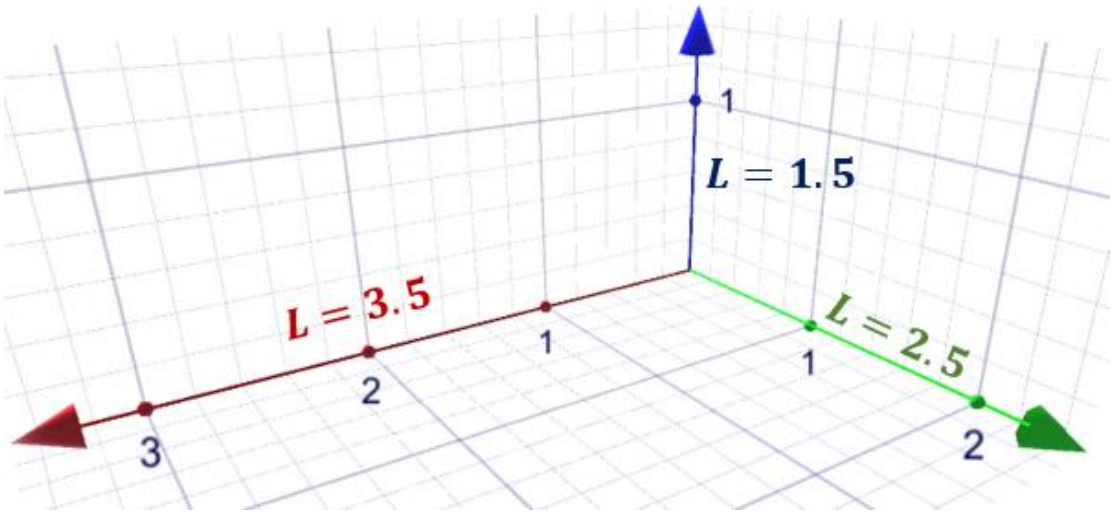
1.2.2.1. Visible: Define la visibilidad del eje.

1.2.2.2. Direction: Define la dirección hacia donde se grafica el eje.

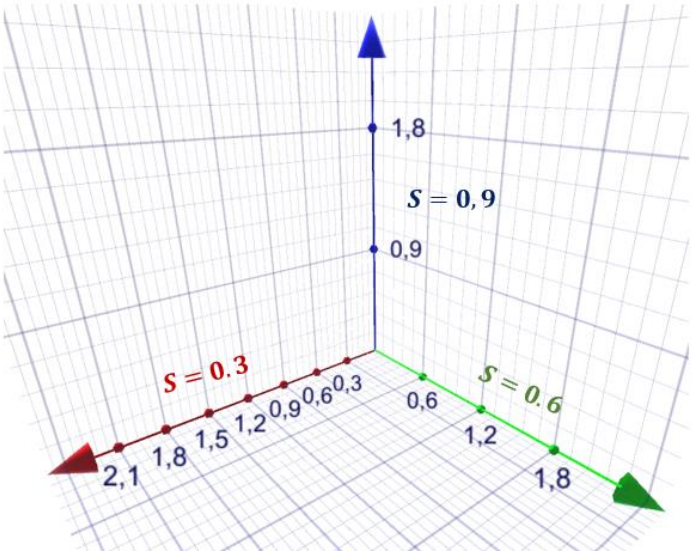


1.2.2.3. Color: Define el color de la línea de eje. Por defecto está configurado *X*: Rojo, *Y*: Verde, *Z* = azul.

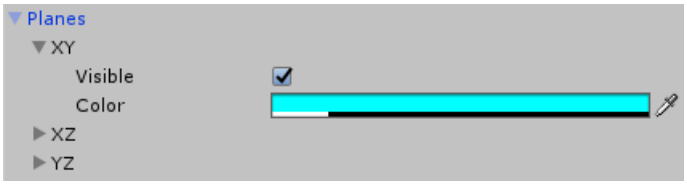
1.2.2.4. Long: Define el largo del eje.



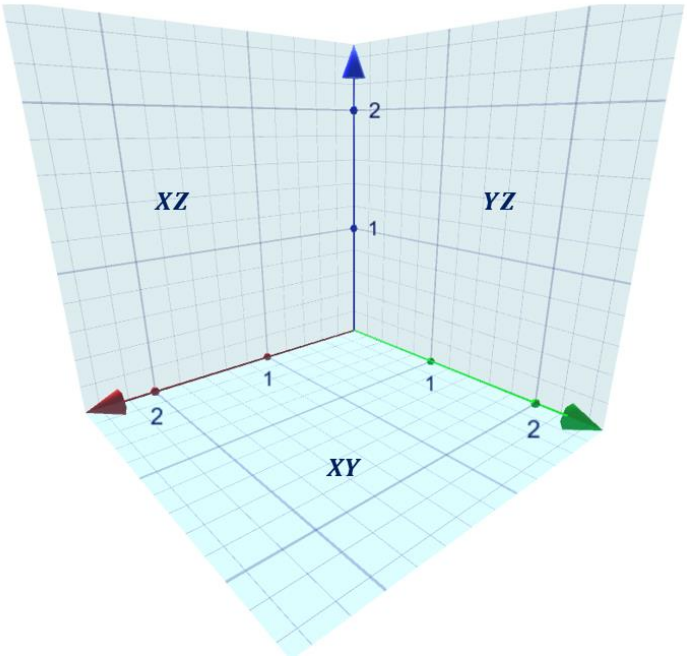
1.2.2.5. Steps: Define que tan separadas están los ticks y las líneas de la grilla.



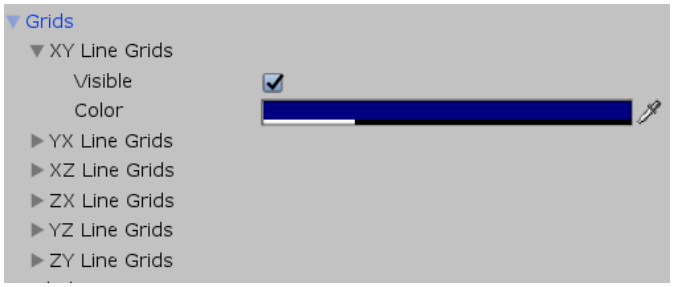
1.2.3. Planes: Define como se renderizan los planos que se generan entre cada una de las líneas de eje. Cada plano *XY, XZ, YZ* se puede manejar por separado, y es personalizable en términos de visibilidad, y color. Todas las demás propiedades del plano como el largo vienen dadas por las propiedades del respectivo eje.

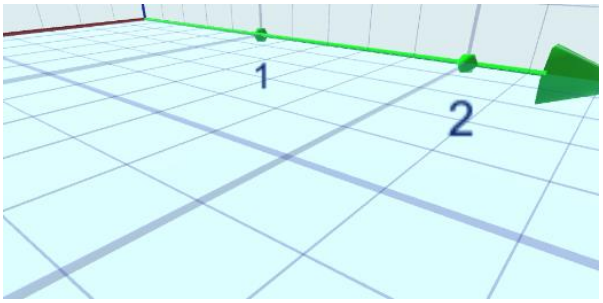


1.2.3.1. Visible: Define la visibilidad del plano.
1.2.3.2. Color: Define el color del plano.



1.2.4. Grids: Define como se renderizan las líneas de grilla que se generan en cada plano. Cada línea de grilla *XY, YX, XZ, ZX, YZ, ZY* se puede manejar por separado, y es personalizable en términos de visibilidad, y color. Todas las demás propiedades de la línea de grilla como el largo vienen dadas por las propiedades del respectivo eje.

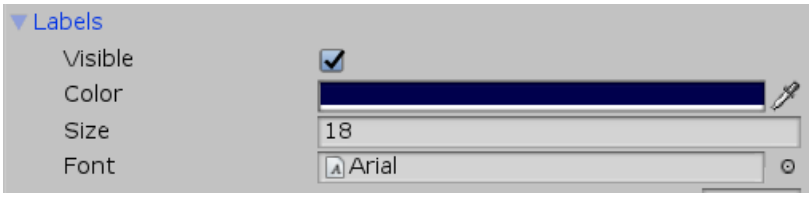




1.2.4.1. Visible: Define la visibilidad de la línea de grilla.

1.2.4.2. Color: Define el color de la línea de grilla.

1.2.5. Labels: Define como se renderizan las etiquetas numeradas que aparecen en cada tick. Estas etiquetas son personalizables globalmente en términos de visibilidad, color, tamaño, y fuente. Las demás propiedades como la frecuencia con la que aparecen y la distancia entre estas vienen dadas por las propiedades del respectivo eje.

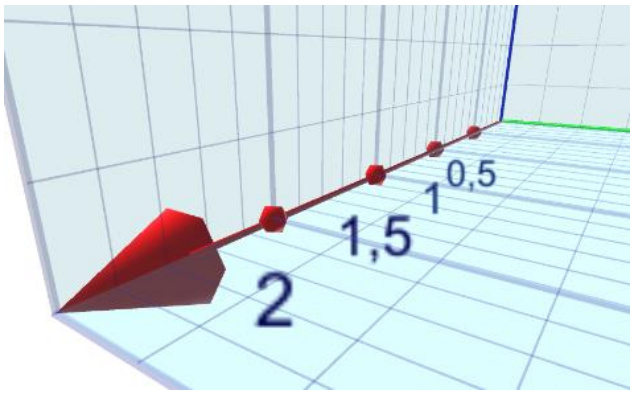


1.2.5.1. Visible: Define la visibilidad de las etiquetas.

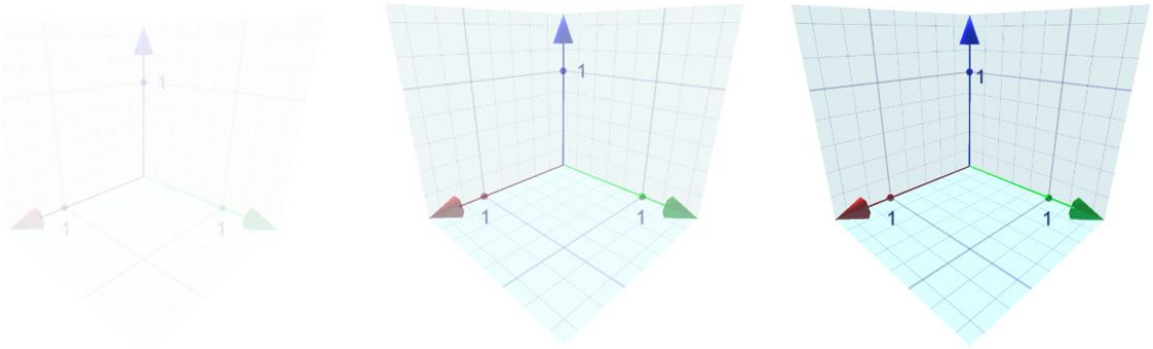
1.2.5.2. Color: Define el color de la fuente de las etiquetas.

1.2.5.3. Size: Define el tamaño de la fuente de las etiquetas.

1.2.5.4. Font: Define la fuente de las etiquetas.



1.2.6. Alpha: Define la transparencia global de los ejes y sus componentes. El parámetro Alpha varía entre 0 y 1, donde 0 es completamente transparente, y 1 es completamente opaco.



$a = 0,1$ $a = 0,5$ $a = 1$

1.3. Agregar vector simple a través de código:

```
Axes axes = Axes.Create();
```

1.4. Métodos públicos:

- **SetMaterial:** Define el material el material que usan todos los modelos del eje.

  Retorna:
  `Null`
  Parámetros:
  `Material material`
  Implementación:
  `SetMaterial(material);`

- **SetArrowModel:** Define el modelo con el que se renderiza la flecha de los ejes.

  Retorna:
  `Null`
  Parámetros:
  `GameObject model`
  Implementación:
  `SetArrowModel(model);`

- **SetArrowRadius:** Define el radio del cuerpo de las líneas de eje.

  Retorna:
  `Null`

Parámetros:

```
float radius
```

Implementación:

```
SetArrowRadius(radius);
```

- **SetArrowLong:** Define el largo de las flechas de los ejes.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Long
  ```

  Implementación:

  ```
  SetArrowLong(Long);
  ```

- **SetAxisRadius:** Define el radio del cuerpo de los ejes.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius;
  ```

  Implementación:

  ```
  SetAxisRadius(radius);
  ```

- **SetTickRadius:** Define el radio de los ticks.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius;
  ```

  Implementación:

  ```
  SetTickRadius(radius);
  ```

- **SetGridLineRadius:** Define el radio de las líneas de la grilla.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius;
  ```

  Implementación:

  ```
  SetGridLineRadius(radius);
  ```

- **SetMinorGridLineFactor:** Define el factor del eje menor. Este factor varía entre 0 y 1, donde 0 es totalmente invisible, y el 1 es el mismo grosor de las líneas de grilla principales.

Retorna:

```
Null
```

Parámetros:

```
float factor;
```

Implementación:

```
SetMinorGridLineFactor(factor);
```

- **SetAxesVisibility:** Define la visibilidad de todas las líneas de eje.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  bool visible
  ```

  Implementación:

  ```
  SetAxisVisibility(visible);
  ```

- **SetAxisVisibility:** Define la visibilidad de un eje en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  Axes.Axis axis
  bool visible

  Implementación:

  ```
  SetAxisVisibility(axis, visible);
  ```

- **SetAxesDirection:** Define la dirección de todas las líneas de eje.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Direction direction;
  ```

  Implementación:

  ```
  SetAxesDirection(direction);
  ```

- **SetAxisDirection:** Define la dirección de una línea de eje en específico**.**

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Axis axis
  Axes.Direction direction
  ```

  Implementación:

  ```
  SetAxisDirection(axis, direction);
  ```

- **SetAxesColor:** Define el color de todas las líneas de eje.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color
  ```

  Implementación:

  ```
  SetAxesColor(color);
  ```

- **SetAxisColor:** Define el color de una línea de eje en específico**.**

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Axis axis
  Color color
  ```

  Implementación:

  ```
  SetAxisColor(axis, color);
  ```

- **SetAxesLong:** Define el largo de todas las líneas de eje.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Long;
  ```

  Implementación:

  ```
  SetAxesLong(Long);
  ```

- **SetAxisLong:** Define el largo de una línea de eje en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Axis axis
  float Long
  ```

  Implementación:

  ```
  SetAxisLong(axis, Long);
  ```

- **SetAxesSteps:** Define que tan separadas están las líneas de la grilla, y los ticks de todos los ejes.

  Retorna:

  ```
  Null
  ```

Parámetros:

```
float steps;
```

Implementación:

```
SetAxisSteps(steps);
```

- **SetAxisSteps:** Define que tan separadas están las líneas de la grilla, y los ticks de un eje en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Axis axis
  float steps
  ```

  Implementación:

  ```
  SetAxisSteps(axis, steps);
  ```

- **SetPlanesVisibility:** Define la visibilidad de todos los planos.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  bool visible;
  ```

  Implementación:

  ```
  SetPlaneVisibility(visible);
  ```

- **SetPlaneVisibility:** Define la visibilidad de un plano en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Plane plane
  Color color;
  ```

  Implementación:

  ```
  SetPlaneVisibility(plane, color);
  ```

- **SetPlanesColor:** Define el color de todos los planos.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color
  ```

  Implementación:

  ```
  SetPlanesColor(color);
  ```

- **SetPlaneColor:** Define el color de un plano en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Plane plane
  Color color
  ```

  Implementación:

  ```
  SetPlaneColor(plane, color);
  ```

- **SetAllGridsVisibility:** Define la visibilidad de todas las líneas de grilla.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  bool visible;
  ```

  Implementación:

  ```
  SetGridsVisibility(visible);
  ```

- **SetGridsVisibility:** Define la visibilidad de una línea de grilla en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Gridlines gridlines
  bool visible
  ```

  Implementación:

  ```
  SetGridsVisibility(gridlines, visible);
  ```

- **SetAllGridsColor:** Define el color de todas las líneas de grilla.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color
  ```

  Implementación:

  ```
  SetAllGridsColor(color);
  ```

- **SetGridsColor:** Define el color de una línea de grilla en específico.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color;
  ```

Implementación:

```
SetGridsColor(gridlines, color);
```

- **SetLabelsVisibility:** Define la visibilidad de las etiquetas.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  bool visible;
  ```

  Implementación:

  ```
  SetLabelsVisibility(visible);
  ```

- **SetLabelsColor:** Define el color de fuente de las etiquetas.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color
  ```

  Implementación:

  ```
  SetLabelsColor(color);
  ```

- **SetLabelSize:** Define el tamaño de fuente de las etiquetas.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  int size
  ```

  Implementación:

  ```
  SetLabelSize(size);
  ```

- **SetLabelsFont:** Define la fuente de las etiquetas.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Font font
  ```

  Implementación:

  ```
  SetLabelsFont(font);
  ```

- **SetAlpha:** Define la transparencia global del eje. Al parámetro Alpha varía entre 0 y 1, donde 0 es completamente transparente, y 1 es completamente opaco.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float alpha;
  ```

Implementación:

```
SetAlpha(alpha);
```

- **GetMaterial**: Retorna el material con el que se renderizan todos los modelos del eje.

  Retorna:

  ```
  Material
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  Material material = axes.GetMaterial();
  ```

- **GetArrowModel:** Retorna el modelo con el que se renderiza la flecha de los ejes.

  Retorna:

  ```
  GameObject
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  GameObject model = axes.GetArrowModel();
  ```

- **GetArrowRadius:** Retorna el radio de las flechas de los ejes.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float radius = axes.GetArrowRadius();
  ```

- **GetArrowLong:** Retorna el largo de las flechas de los ejes.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float Long = axes.GetArrowLong();
  ```

- **GetAxisRadius:** Retorna el radio del cuerpo de las líneas de eje.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float axisRadius = axes.GetAxisRadius();
  ```

- **GetTickRadius:** Retorna el radio de los ticks de las líneas de eje.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float radius = axes.GetTickRadius();
  ```

- **GetGridLineRadius:** Retorna el radio de las líneas de grilla.

  Retorna:
  ```
  float;
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float radius = axes.GetGridLineRadius();
  ```

- **GetMinorGridLineFactor:** Retorna el factor de las líneas de grilla secundarias.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float factor = axes.GetMinorGridLineFactor();
  ```

- **GetAxesVisibility:** Retorna el estado de visualización de todos los ejes.

  Retorna:
  ```
  bool
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  bool visible = axes.GetAxesVisibility();
  ```

- **GetAxisVisibility:** Retorna el estado de visualización de un eje en particular.

  Retorna:
  ```
  bool
  ```
  Parámetros:
  ```
  Axes.Axis axis
  ```
  Implementación:
  ```
  bool visible = axes.GetAxesVisibility(axis);
  ```

- **GetAxisDirection:** Retorna la dirección de un eje en particular.

  Retorna:

  ```
  Direction
  ```

  Parámetros:

  ```
  Axes.Axis axis;
  ```

  Implementación:

  ```
  Direction direction = axes.GetAxisDirection(axis);
  ```

- **GetAxisColor:** Retorna el color de un eje en particular.

  Retorna:

  ```
  Color
  ```

  Parámetros:

  ```
  Axes.Axis axis;
  ```

  Implementación:

  ```
  Color color = axes.GetAxisColor(axis);
  ```

- **GetAxisLong:** Retorna el largo de un eje en particular.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Axes.Axis axis;
  ```

  Implementación:

  ```
  float Long = axes.GetAxisLong(axis);
  ```

- **GetAxisSteps:** Retorna los pasos de un eje en particular.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Axes.Axis axis;
  ```

  Implementación:

  ```
  float steps = axes.GetAxisSteps(axis);
  ```

- **GetPlaneVisibility:** Retorna es estado de visibilidad de un plano en particular.

  Retorna:

  ```
  bool
  ```

  Parámetros:

  ```
  Axes.Plane plane
  ```

  Implementación:

  ```
  bool visible = axes.GetPlaneVisibility(plane);
  ```

- **GetPlaneColor:** Retorna el color de un plano en particular.

  Retorna:

  ```
  Color
  ```

  Parámetros:

  ```
  Axes.Plane plane
  ```

  Implementación:

  ```
  Color color = axes.GetPlaneColor(plane);
  ```

- **GetGridsVisibility:** Retorna el estado de visibilidad de una línea de grilla en particular. Si el parámetro *gridlines* es *null* se retorna el estado de visibilidad de todas las grillas.

  Retorna:

  ```
  bool
  ```

  Parámetros:

  ```
  Axes.Gridlines gridlines (opcional)
  ```

  Implementación:

  ```
  bool visible = axes.GetGridsVisibility(gridlines);
  ```

- **GetGridsColor:** Retorna el color de una línea de grilla en particular.

  Retorna:

  ```
  Color
  ```

  Parámetros:

  ```
  Axes.Gridlines gridlines
  ```

  Implementación:

  ```
  Color color = axes.GetGridsColor(gridlines);
  ```

- **GetLabelsVisibility:** Retorna el estado de visibilidad de las etiquetas numeradas

  Retorna:

  ```
  bool
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  bool visible = axes.GetLabelsVisibility();
  ```

- **GetLabelsColor:** Retorna el color de la fuente de las etiquetas numeradas.

  Retorna:

  ```
  Color
  ```

  Parámetros:

  ```
  Null
  ```

Implementación:

```
Color color = axes.GetLabelsColor();
```

- **GetLabelsSize:** Retorna el tamaño de la fuente de las etiquetas numeradas.

  Retorna:

  ```
  int
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  int size = axes.GetLabelSize();
  ```

- **GetLabelsFont:** Retorna la fuente usada para las etiquetas numeradas.

  Retorna:

  ```
  Font
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  Font font = axes.GetLabelsFont();
  ```

- **GetAlpha:** Retorna la transparencia global de todo el sistema de ejes.

  Retorna:

  ```
  float
  ```

  Parámetros:

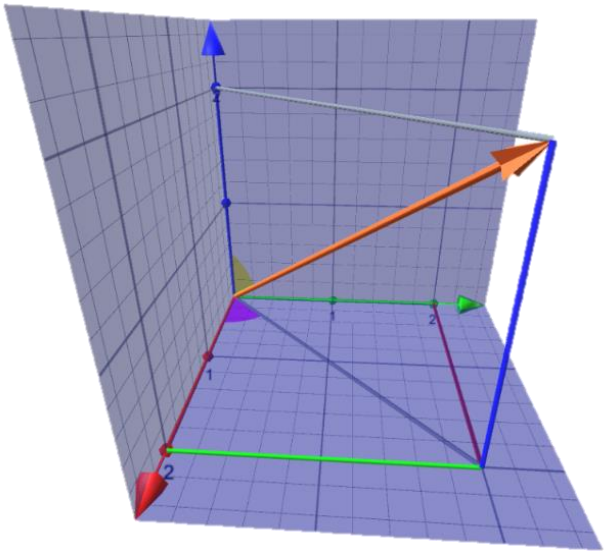
  ```
  Null
  ```

  Implementación:

  ```
  float alpha = axes.GetAlpha();
  ```

2. Graficar vectores:

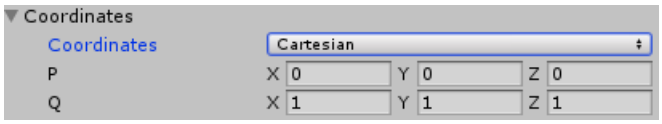
R3+ permite graficar vectores simples de una manera muy dinámica, sencilla, y personalizable tanto por código como por inspector. Los vectores se pueden apuntar en coordenadas cartesianas, cilíndricas y esféricas, y son totalmente personalizables en términos de color, modelo, norma de renderizado y modelo cuando la norma vale cero.



2.1. Agregar un vector simple a través de inspector:

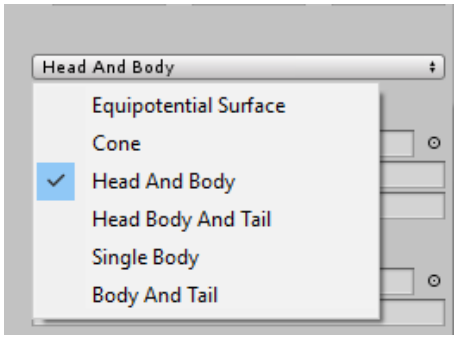
I. Crear un nuevo *GameObject*
II. Agregar el componente "*Vector*"

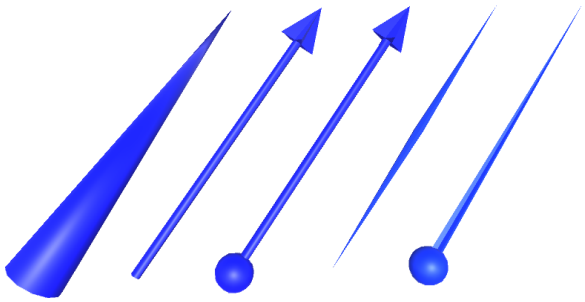
2.2. Modificar un vector simple a través de inspector:



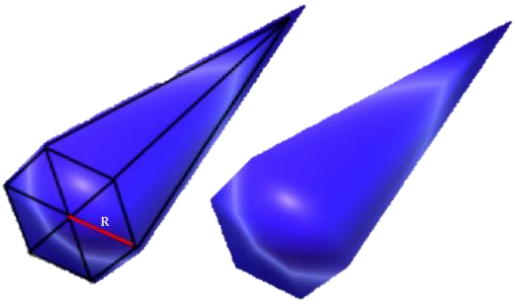
2.2.1. Coordenadas: Define la posición donde inicia el vector **P** en coordenadas cartesianas, y la dirección **Q** hacia donde apunta.
El tipo de coordenadas que maneja **Q** se puede especificar en la lista desplegable **coordinates** de 3 maneras: cartesianas (x, y, z), cilíndricas (ρ, Φ, z) o esféricas (r, θ, Φ)

2.2.2. Vector Model: Define la forma en que se va a renderizar el vector en la escena. Se puede personalizar el modelo del vector simple y el modelo del vector cuando la norma vale cero por separado. Hay 5 formas de renderizar un vector simple, cada una con sus propias maneras de personalizar el modelo, y una forma de graficar un diferencial de superficie equipotencial.



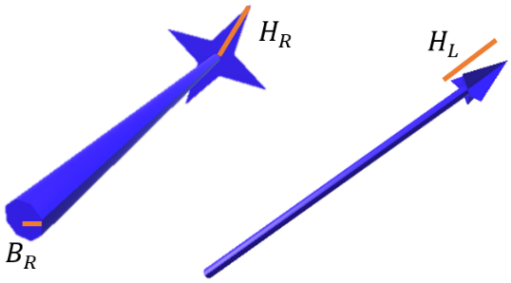


2.2.2.1. Cone: Es la forma más eficiente de hacer la gráfica de un vector, pues no carga un mesh de forma externa, y utiliza un número reducido de polígonos que son calculados a través de un algoritmo. Este modelo permite personalizar:



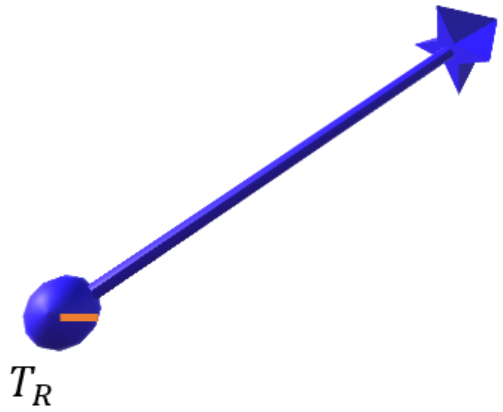
- **Radius:** Radio del cono
- **Divs:** Número de polígonos que tendrá la base del cono

2.2.2.2. Head And Body: Carga dos mesh independientes, uno para el cuerpo y uno para la cabeza. Ambos mesh se pueden modificar por separado, así como sus propiedades. Este modelo permite personalizar:



- **Head Model:** Mesh que se utiliza para renderizar la cabeza del vector
- **Head Radius:** Radio de la cabeza
- **Head Long:** Largo de la cabeza
- **Body Model:** Mesh que se utiliza para renderizar el cuerpo del vector
- **Body Radius:** Radio del cuerpo

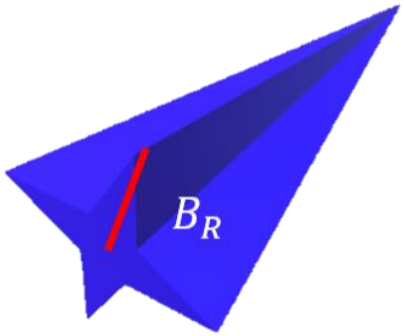
2.2.2.3. Head Body And Tail: Carga tres mesh independientes, para la cabeza, el cuerpo y la cola. Todos los mesh pueden ser modificados por separado, así como sus propiedades. Este modelo permite personalizar:



- **Head Model:** Mesh que se utiliza para renderizar la cabeza del vector
- **Head Radius:** Radio de la cabeza
- **Head Long:** Largo de la cabeza
- **Body Model:** Mesh que se utiliza para renderizar el cuerpo del vector
- **Body Radius:** Radio del cuerpo
- **Tail Model:** Mesh que se utiliza para renderizar la cola del vector
- **Tail Radius**: Radio de la cola

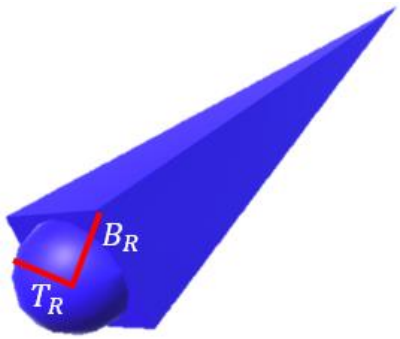
2.2.2.4. Single Body: Carga un solo mesh, que deforma su largo en función de la norma del vector. Este modelo permite personalizar:

- **Body Model:** Mesh que se utiliza para renderizar el cuerpo del vector
- **Body Radius:** Radio del cuerpo



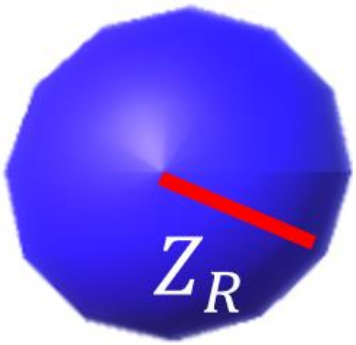
2.2.2.5. Body and Tail: Carga dos mesh independientes, uno para el cuerpo y uno para la cola. Ambos mesh se pueden cambiar por separado, así como sus propiedades. Este modelo permite personalizar:

- **Body Model:** Mesh que se utiliza para renderizar el cuerpo del vector
- **Body Radius:** Radio del cuerpo
- **Tail Model:** Mesh que se utiliza para renderizar la cabeza del vector
- **Tail Radius:** Radio de la cabeza
- **Tail Long:** Largo de la cabeza

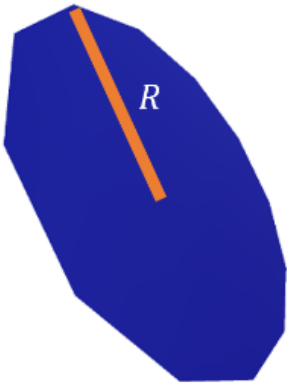


2.2.2.6. Zero: Es el modelo que se cargará cuando el vector tiene una norma de 0. Este modelo permite personalizar:

- **Zero Model:** Mesh que se utiliza para renderizar el vector de norma cero.
- **Zero Radius:** Radio del mesh.

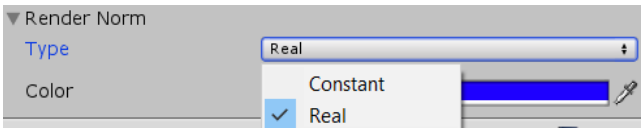


2.2.2.7. Equipotencial surface: Grafica un diferencial circular de una superficie equipotencial. Este modelo permite personalizar:

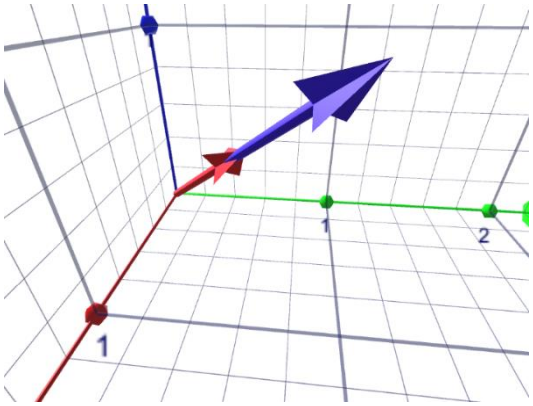


- **Radius:** Radio del diferencial de superficie equipotencial

2.2.3. Render Norm: Define el tipo de norma con la cual se va a renderizar el vector. Hay 2 formas de renderizar:



- **Real**: El vector se va a renderizar con el tamaño real que debe tener según sus componentes
- **Constant**: El vector se va a renderizar con un tamaño constante, aunque apuntará en la misma dirección



Aquí vemos dos vectores que apuntan a la dirección [1,1,1].

El azul tiene norma real, y el rojo tiene norma constante = 1.

2.2.4. Color: Define el color del vector.

2.3. Agregar vector simple a través de código:

```
Vector vector = Vector.Create("nombreDelVector");
```

2.4. Métodos públicos:

- **SetP:** Define la posición en coordenadas cartesianas donde iniciará el vector**.**

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Vector3 P
  ```

  Implementación:

  ```
  SetP(P);
  ```

- **SetQ:** Define hacia donde apunta el vector. Esta dirección puede venir dada en coordenadas cartesianas, cilíndricas o esféricas, dependiendo del segundo parámetro. Si el segundo parámetro es null, se toma por defecto coordenadas cartesianas.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Vector3 Q
  MathV.Coordinates coordinates (Opcional)
  ```

  Implementación:

  ```
  SetQ(Q, coordinates);
  ```

- **SetVectorModelType:** Define el tipo de modelo que utilizará el vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  VectorModel.Type type
  ```

  Implementación:

  ```
  SetVectorModelType(type);
  ```

- **SetConeDivs:** Define cuantas divisiones va a tener la base del cono (En caso se ser el modelo de cuerpo seleccionado)**.**

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  int divs
  ```

  Implementación:

  ```
  SetConeDivs(divs);
  ```

- **SetHeadModel:** Define el modelo con el que se renderiza la cabeza del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetHeadModel(model);
  ```

- **SetHeadRadius:** Define el radio con el que se renderiza la cabeza del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetHeadRadius(radius);
  ```

- **SetHeadLong:** Define el largo con el que se renderiza la cabeza del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Long
  ```

  Implementación:

  ```
  SetHeadLong(Long);
  ```

- **SetBodyModel:** Define el modelo con el que se renderiza el cuerpo del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetBodyModel(model);
  ```

- **SetBodyRadius:** Define el radio con el que se renderiza el cuerpo del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetBodyRadius(radius);
  ```

- **SetTailModel:** Define el modelo con el que se renderiza la cola del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetTailModel(model);
  ```

- **SetTailRadius:** Define el radio con el que se renderiza la cola del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetTailRadius(radius);
  ```

- **SetZeroModel:** Define el modelo con el que se renderiza el vector cuando su norma vale cero.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetTailModel(model);
  ```

- **SetZeroRadius:** Define el radio con el que se renderiza el vector cuando su norma vale cero.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetTailRadius(radius);
  ```

- **SetRenderNormType:** Define la forma en que se renderiza la norma del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  VectorProps.RenderNormSV.Type type
  ```

Implementación:

```
SetRenderNormType(type);
```

- **SetRenderNormConstant:** Define el largo con el que se renderiza el vector si el type es *Constant*. En caso de ser *Real,* el *type* es cambiado automáticamente a *Constant.*

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float norm;
  ```

  Implementación:

  ```
  SetRenderNormConstant(norm);
  ```

- **SetColor:** Define el color del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color;
  ```

  Implementación:

  ```
  SetColor(color);
  ```

- **GetCoordinates:** Retorna el tipo de coordenadas que maneja el vector.

  Retorna:

  ```
  MathV.Coordinates
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  MathV.Coordinates coordinates = vector.GetCoordinates();
  ```

- **GetP:** Retorna la coordenada donde nace el vector. En caso de no especificarse el tipo de coordenada deseada como parámetro, se retornará el valor en coordenadas cartesianas.

  Retorna:

  ```
  Vector3
  ```

  Parámetros:

  ```
  MathV.Coordinates coordinates (Opcional)
  ```

  Implementación:

  ```
  Vector3 P = vector.GetP(coordinates);
  ```

- **GetQ:** Retorna la coordenada a donde apunta el vector. En caso de no especificarse el tipo de coordenada deseada como parámetro, se retornará el valor en coordenadas cartesianas.

  Retorna:
  ```
  Vector3
  ```
  Parámetros:
  ```
  MathV.Coordinates coordinates (Opcional)
  ```
  Implementación:
  ```
  Vector3 Q = vector.GetQ(coordinates);
  ```

- **GetX:** Retorna la coordenada X del vector.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float X = vector.GetX();
  ```

- **GetY:** Retorna la coordenada Y del vector.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float Y = vector.GetY();
  ```

- **GetZ:** Retorna la coordenada Z del vector.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  float Z = vector.GetZ();
  ```

- **GetNorm:** Retorna el valor de la norma del vector.

  Retorna:
  ```
  float
  ```
  Parámetros:
  ```
  Null
  ```

Implementación:

```
float norm = vector.GetNorm();
```

- **GetRo:** Retorna la magnitud de la proyección del vector en el plano *xy*

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float ro = vector.GetRo();
  ```

- **GetTheta:** Retorna el ángulo θ de coordenadas esféricas.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float theta = vector.GetTheta();
  ```

- **GetPhi:** Retorna el ángulo ϕ de coordenadas esféricas.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float phi = vector.GetPhi();
  ```

- **GetModelType:** Retorna el tipo de modelo que usa el vector.

  Retorna:

  ```
  VectorModel.Type
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  VectorModel.Type type = vector.GetType();
  ```

- **GetConeDivs:** Retorna el número de divisiones que tiene el modelo cono.

  Retorna:

  ```
  int
  ```

  Parámetros:

  ```
  Null
  ```

Implementación:

```
int divs = vector.GetConeDivs();
```

- **GetHeadModel:** Retorna el modelo de la cabeza del vector.

  Retorna:

  ```
  GameObject
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  GameObject model = vector.GetHeadModel();
  ```

- **GetHeadRadius:** Retorna el radio de la cabeza del vector.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float radius = vector.GetHeadRadius();
  ```

- **GetHeadLong:** Retorna el largo de la cabeza del vector.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float Long = vector.GetHeadLong();
  ```

- **GetBodyModel:** Retorna el modelo del cuerpo del vector.

  Retorna:

  ```
  GameObject
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  GameObject model = vector.GetBodyModel();
  ```

- **GetBodyRadius:** Retorna el radio del cuerpo del vector.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

Implementación:

```
float radius = vector.GetBodyRadius();
```

- **GetTailModel:** Retorna el modelo de la cola del vector.

  Retorna:

  `GameObject`

  Parámetros:

  `Null`

  Implementación:

  ```
  GameObject model = vector.GetTailModel();
  ```

- **GetTailRadius:** Retorna el radio del modelo de la cola del vector.

  Retorna:

  `float`

  Parámetros:

  `Null`

  Implementación:

  ```
  float radius = vector.GetTailRadius();
  ```

- **GetZeroModel:** Define el modelo con el que se renderiza el vector cuando su norma vale cero.

  Retorna:

  `GameObject`

  Parámetros:

  `Null`

  Implementación:

  ```
  GameObject model = vector.GetZeroModel();
  ```

- **GetZeroRadius:** Retorna el radio con el que se renderiza el vector cuando su norma vale cero.

  Retorna:

  `float`

  Parámetros:

  `Null`

  Implementación:

  ```
  float radius = vector.GetZeroRadius();
  ```

- **GetRenderNormType:** Retorna el tipo de norma con la que se está renderizando el vector.

Retorna:

`VectorProps.RenderNormSV.Type`

Parámetros:

`Null`

Implementación:

```
VectorProps.RenderNormSV.Type t=vector.GetRenderNormType();
```

- **GetRenderNormValue:** Retorna el valor de la norma con el que se está renderizando el vector.

  Retorna:

  `float`

  Parámetros:

  `Null`

  Implementación:

  ```
  float renderNormValue = vector.GetRenderNormValue();
  ```

- **GetColor:** Retorna el color del vector**.**

  Retorna:

  `Color`

  Parámetros:

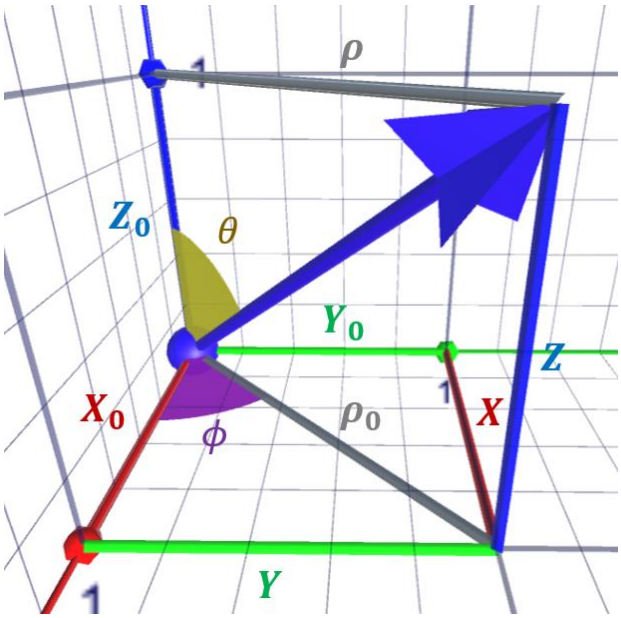
  `Null`

  Implementación:

  ```
  Color color = vector.GetColor();
  ```

3. Graficar proyecciones de vectores:

R3+ permite graficar proyecciones de vectores simples de una manera dinámica, sencilla, y personalizable tanto por código como por inspector. Las proyecciones se pueden manejar y personalizar por separado y se actualizan dinámicamente, al tiempo que el vector lo hace.



Las proyecciones que se pueden graficar son: *X0, X, Y0, Y, Z0, Z, ρ0, ρ, θ, ϕ*

3.1. Agregar proyecciones a un vector simple a través de inspector:

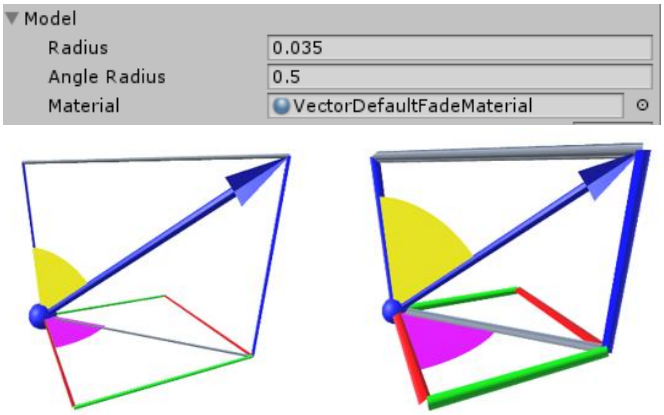
I. Crear un nuevo *GameObject*, o tomar un *GameObject* que ya tenga el componente "*Vector*"
II. Agregar el componente "*Projections*"

3.2. Modificar proyecciones de un vector simple a través de inspector: cada una de las proyecciones se pueden manejar por separado, así como personalizar en términos de color y visibilidad.

3.2.1. Projections: Define como se renderizan las proyecciones individuales. Cada proyección *X0, X, Y0, Y, Z0, Z, ρ0, ρ, θ, ϕ* se puede manejar por separado, y es personalizable en términos de visibilidad y color.

3.2.1.1. Visible: Define la visibilidad del eje.
3.2.1.2. Direction: Define la dirección hacia donde se grafica el eje.

3.2.2. Model: Define como se renderizan todas las proyecciones al tiempo en términos de grosor de proyección, radio de las proyecciones angulares y material.

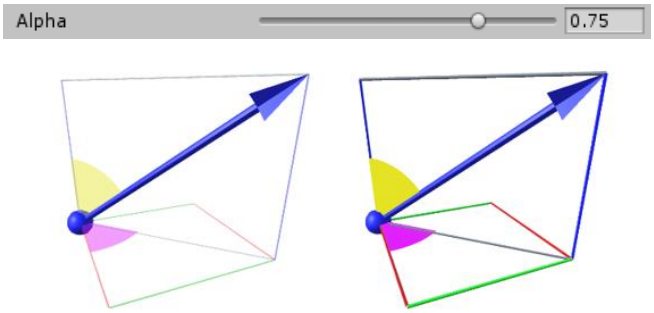


3.2.2.1. Radius: Define el grosor de las proyecciones.

3.2.2.2. Angle Radius: Define el radio de las proyecciones angulares.

3.2.2.3. Material: Material que tendrán todos los modelos que conforman la proyección.

3.2.3. Alpha: Permite personalizar la transparencia de todas las proyecciones al mismo tiempo.



3.3. Agregar vector simple a través de código:

```
Projections projections = vector.AddProjections();
```

3.4. Métodos públicos:

- **SetProjectionsVisibility:** Permite definir que proyecciones serán visibles en función del espacio que se desee.

  Retorna:

  `Null`

  Parámetros:

  `MathV.Coordinates coordinates`

  Implementación:

  `SetProjectionsVisibility(coordinates);`

- **SetProjectionsVisibility:** Permite definir que proyecciones serán visibles por separado.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Projections.projection projection
  bool state
  ```

  Implementación:

  ```
  SetProjectionsVisibility(projectionType, state);
  ```

- **SetModelRadius:** Define el grosor que tendrán las proyecciones.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Radius
  ```

  Implementación:

  ```
  SetModelRadius(radius);
  ```

- **SetAngleRadius:** Define el radio que tendrán las proyecciones angulares.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Radius
  ```

  Implementación:

  ```
  SetAngleRadius(radius);
  ```

- **SetAlpha:** Define la transparencia que tendrán todas las proyecciones.

  Retorna:

  ```
  Null
  ```

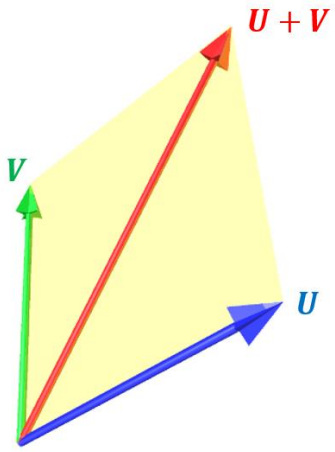
  Parámetros:

  ```
  float Alpha
  ```

  Implementación:

  ```
  SetAlpha(alpha);
  ```

4. Graficar suma de vectores:

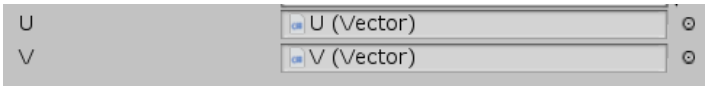


R3+ permite graficar el vector suma entre dos vectores. Este vector suma es dinámico, personalizable, y se actualiza automáticamente en función de los vectores sumados.

4.1. Agregar un vector suma a través de inspector:

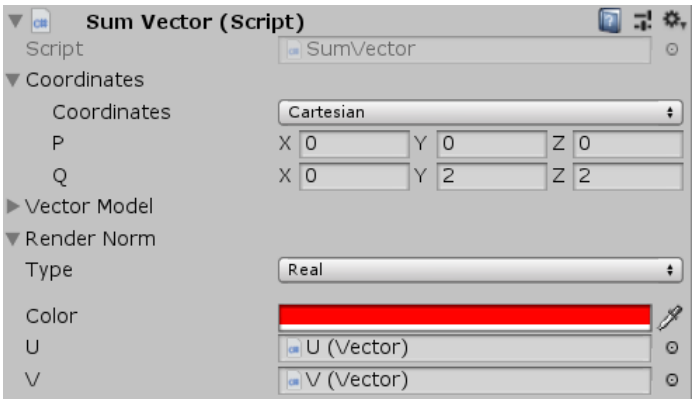
I. Crear un nuevo *GameObject*
II. Agregar el componente "*Sum Vecto*r"

Para que el vector suma sea visible hay que agregar los dos vectores sumandos a través del inspector; esto se hace en los campos *U* y *V*. Una vez agregados, el vector suma se graficará y actualizará automáticamente cuando la coordenada **Q** de cualquiera de los dos sumandos se modifique. Para que el vector suma se pueda graficar es necesario que la coordenada **P** de ambos vectores sumandos sea igual.



4.2. Modificar vector suma a través de inspector:

Sum Vector es una clase que hereda todos los métodos de *Vector*, por lo tanto, maneja los mismos métodos y funciones que este. Lo único que agrega es que se puede cambiar los vectores sumandos, al cambiar el contenido de *U* y *V*.



4.3. Agregar vector suma a través de código:

```
SumVector UpV = SumVector.Create(U, V);
```

4.4. Métodos públicos:

- Todos los métodos de *Vector*

- **SetVectors:** Permite definir los vectores sumandos, que conforman el vector suma.

  Retorna:

  `Null`

Parámetros:

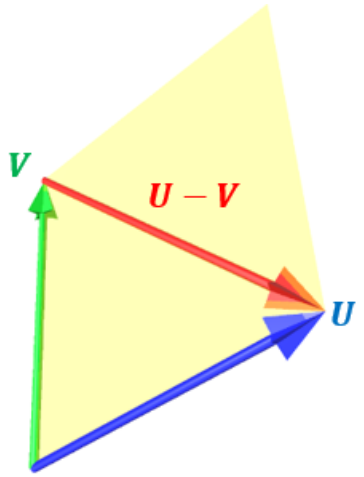
Vector U
Vector V

Implementación:

```
SetVectors(U, V);
```

5. Graficar resta de vectores:

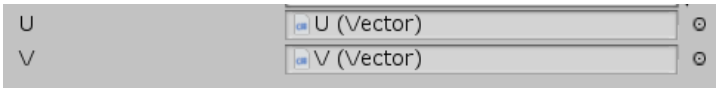
R3+ permite graficar el vector resta entre dos vectores. Este vector resta es dinámico, personalizable, y se actualiza automáticamente en función de los vectores restados.



5.1. Agregar un vector resta a través de inspector:

I. Crear un nuevo *GameObject*
II. Agregar el componente "*Diff Vector*"

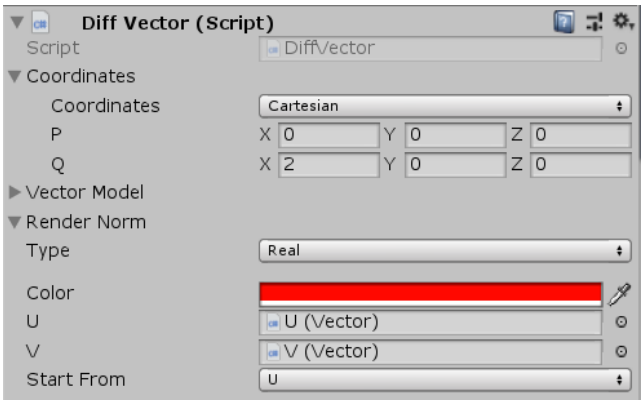
Para que el vector resta sea visible hay que agregar los dos vectores a restar a través del inspector; esto se hace en los campos *U* y *V*. Una vez agregados, el vector resta se graficará y actualizará automáticamente cuando la coordenada **Q** de cualquiera de los vectores restados se modifique. Para que el vector resta se pueda graficar es necesario que la coordenada **P** de ambos vectores sumandos sea igual.



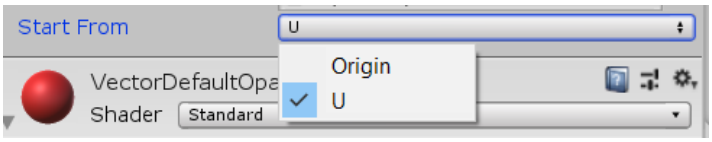
La resta de los vectores siempre será el vector introducido en el campo *U* (minuendo) menos el vector introducido en el campo *V* (sustraendo)

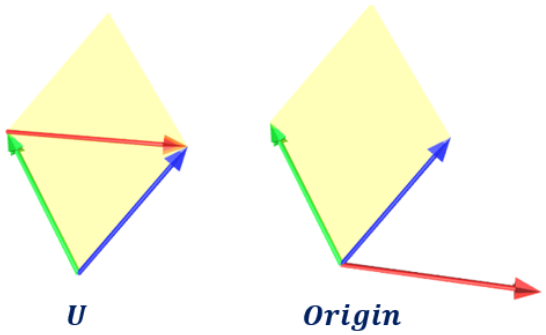
5.2. Modificar vector resta a través de inspector:

Diff Vector es una clase que hereda todos los métodos de *Vector*, por lo tanto, maneja los mismos métodos y funciones que este. Lo único que agrega es que se puede cambiar la coordenada desde donde inicia la gráfica del vector, y los vectores restados, al cambiar el contenido de *U* y *V*.



5.2.1. Start From: Define en donde debe ser el origen del vector resta. Se permite que el origen sea en el mismo origen de los dos vectores, o en donde termina el vector U.





5.3. Agregar vector resta a través de código:

```
DiffVector UdV = DiffVector.Create(U, V);
```

5.4. Métodos públicos:

- Todos los métodos de *Vector*

- **SetVectors:** Permite definir los vectores minuendo y sustraendo, que conforman el vector resta.

  Retorna:
  `Null`
  Parámetros:
  `Vector` `U`
  `Vector` `V`
  Implementación:
  `SetVectors(U, V);`

- **SetStartFrom:** Permite definir en donde iniciará el vector resta.

  Retorna:
  `Null`
  Parámetros:
  `DiffVector.StartFrom` `startFrom`
  Implementación:
  `SetStartFrom (startFrom);`

- **GetStartFrom:** Retorna en donde iniciará el vector resta.

  Retorna:

  `DiffVector.StartFrom`

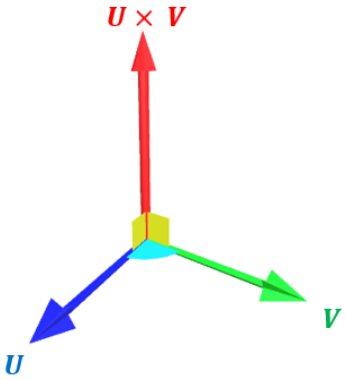
  Parámetros:

  `Null`

  Implementación:

  ```
  DiffVector.StartFrom startFrom = GetStartFrom();
  ```

6. Graficar producto cruz entre vectores:

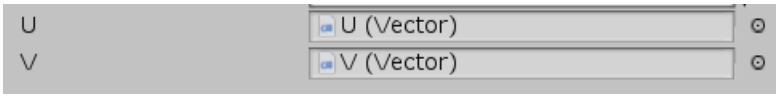


R3+ permite graficar el vector producto cruz entre dos vectores. El vector resultante es dinámico, personalizable, y se actualiza automáticamente en función de los vectores operados.

6.1. Agregar un vector producto cruz a través de inspector:

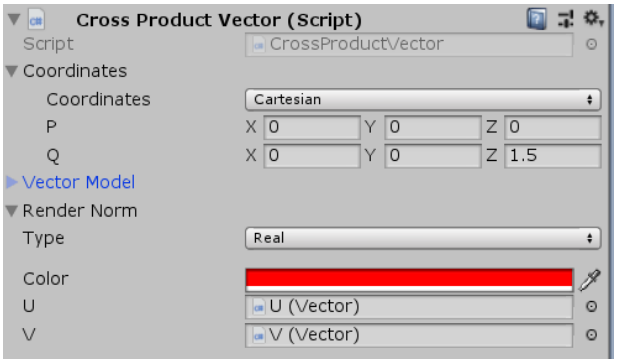
I. Crear un nuevo *GameObject*
II. Agregar el componente "*Cross Product Vecto*r"

Para que el vector producto cruz sea visible hay que agregar los dos vectores a operar a través del inspector; esto se hace en los campos *U* y *V*. Una vez agregados, el vector producto cruz se graficará y actualizará automáticamente cuando la coordenada **Q** de cualquiera de los dos vectores operados se modifique. Para que el vector producto cruz se pueda graficar es necesario que la coordenada **P** de ambos vectores operados sea la misma.



6.2. Modificar vector producto cruz a través de inspector:

Cross Product Vector es una clase que hereda todos los métodos de *Vector*, por lo tanto, maneja los mismos métodos y funciones que este. Lo único que agrega es que se puede cambiar los vectores operados, al cambiar el contenido de *U* y *V*.



6.3. Agregar vector suma a través de código:

```
CrossProductVector UxV = CrossProductVector.Create(U, V);
```

6.4. Métodos públicos:

- Todos los métodos de *Vector*

- **SetVectors:** Permite definir los vectores operados para generar el vector producto cruz.

Retorna:

Null

Parámetros:

Vector U

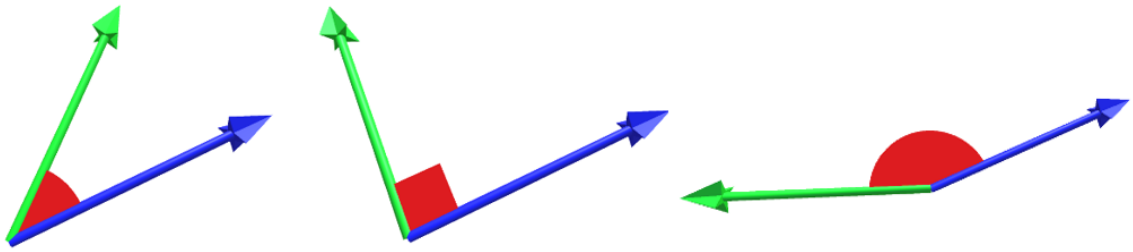
Vector V

Implementación:

```
SetVectors(U, V);
```

7. Graficar ángulo entre dos vectores:

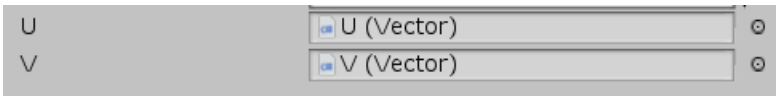
R3+ permite graficar el ángulo entre dos vectores. Este ángulo es dinámico, personalizable, y se actualiza automáticamente en función de los vectores que lo generan.



Cuando el ángulo entre dos vectores es de 90°, se grafica un cuadrado; cuando es diferente de 90 se grafica un segmento de circunferencia. El ángulo graficado siempre es menor a 180°.

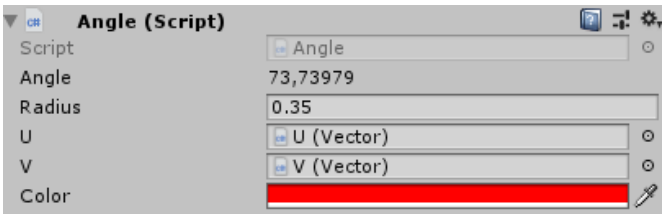
7.1. Agregar un ángulo a través de inspector:

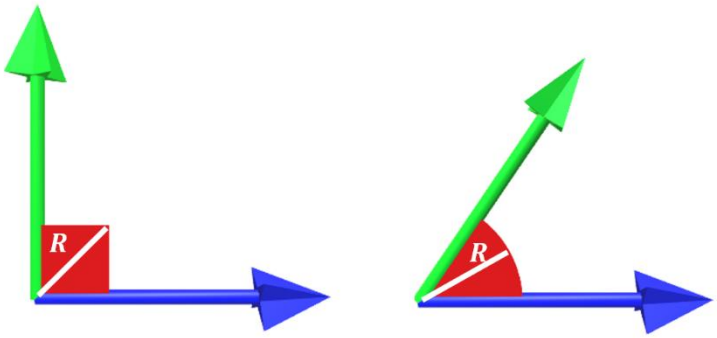
I. Crear un nuevo *GameObject*
II. Agregar el componente "*Angle*"



Para que el ángulo sea visible hay que agregar los dos vectores que lo generan a través del inspector; esto se hace en los campos *U* y *V*. Una vez agregados el ángulo se graficará y actualizará automáticamente cuando la coordenada **Q** de cualquiera de los dos vectores operados se modifique. Para que el ángulo se pueda graficar es necesario que la coordenada **P** de ambos vectores operados sea la misma.

7.2. Modificar ángulo entre dos vectores a través de inspector: La forma en que se grafica el ángulo es personalizable en términos de color y radio. El parámetro *Angle* no es modificable y sirve para indicar la magnitud del ángulo que está siendo graficado.





7.2.1. Radius: Permite personalizar el radio del segmento de círculo que representa el ángulo. Cuando el ángulo es de 90° se personaliza la diagonal del cuadrado que representa el ángulo recto.

7.2.2. Color: Define el color del ángulo

7.3. Agregar ángulo a través de código:

```
Angle angle = Angle.Create(U, V);
```

7.4. Métodos públicos:

- **SetVectors:** Permite definir los vectores que generan el ángulo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Vector U
  Vector V
  ```

  Implementación:

  ```
  SetVectors(U, V);
  ```

- **SetRadius:** Define el radio del segmento de círculo que representa el ángulo. Cuando el ángulo es de 90° se personaliza la diagonal del cuadrado que representa el ángulo recto.

Retorna:

```
Null
```

Parámetros:

```
float Radius
```

Implementación:

```
SetRadius(radius);
```

- **SetColor:** Define el color del ángulo.

  Implementación:

  ```
  SetColor(color);
  ```

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color;
  ```

- **GetAngle:** Retorna el valor del ángulo en grados.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float angle = angle.GetAngle();
  ```

- **GetRadius:** Retorna el radio del segmento de círculo que representa el ángulo. Cuando el ángulo es de 90° retorna la diagonal del cuadrado que representa el ángulo recto**.**

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float radius = angle.GetRadius();
  ```

- **GetColor:** Retorna el color del ángulo.

  Retorna:

  ```
  Color
  ```

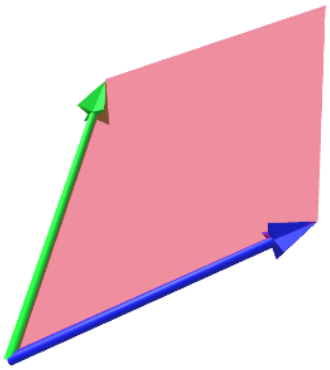
  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  Color color = angle.GetColor();
  ```

8. Graficar paralelogramo entre dos vectores:

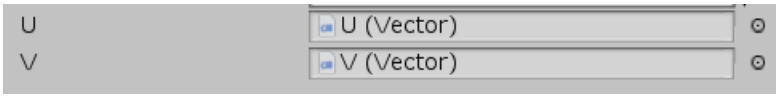


R3+ permite graficar el paralelogramo que se forma entre dos vectores. Este ángulo es dinámico, personalizable, y se actualiza automáticamente en función de los vectores que lo generan.

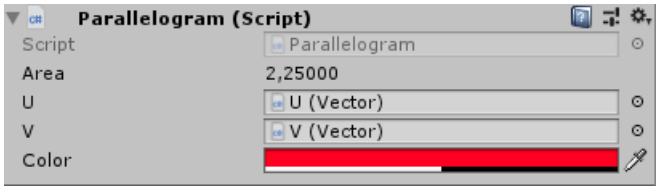
8.1. Agregar un paralelogramo a través de inspector:

I. Crear un nuevo *GameObject*
II. Agregar el componente "*Parallelogram*"

Para que el paralelogramo sea visible hay que agregar los dos vectores que lo generan a través del inspector; esto se hace en los campos U y V. Una vez agregados el paralelogramo se graficará y actualizará automáticamente cuando la coordenada **Q** de cualquiera de los dos vectores operados se modifique. Para que el paralelogramo se pueda graficar es necesario que la coordenada **P** de ambos vectores operados sea la misma.



8.2. Modificar paralelogramo a través de inspector:



La forma en que se grafica el paralelogramo es personalizable en términos de color El parámetro *Area* no es modificable y sirve para indicar el área del paralelogramo que está siendo graficado.

8.2.1. Color:

Define el color del paralelogramo

8.3. Agregar paralelogramo a través de código:

```
Parallelogram parallelogram = Parallelogram.Create(U, V);
```

8.4. Métodos públicos:

- **SetVectors:** Permite definir los vectores que generan el paralelogramo.

  Retorna:

  `Null`

  Parámetros:

Vector U
Vector V

Implementación:

```
SetVectors(U, V);
```

- **SetColor:** Define el color del paralelogramo.

  Retorna:

  Null

  Parámetros:

  Color color;

  Implementación:

  ```
  SetColor(color);
  ```

- **GetArea:** Retorna el valor del área del paralelogramo.

  Retorna:

  float

  Parámetros:

  Null

  Implementación:

  ```
  float area = parallelogram.GetArea();
  ```

- **GetColor:** Retorna el color del paralelogramo.

  Retorna:

  Color

  Parámetros:

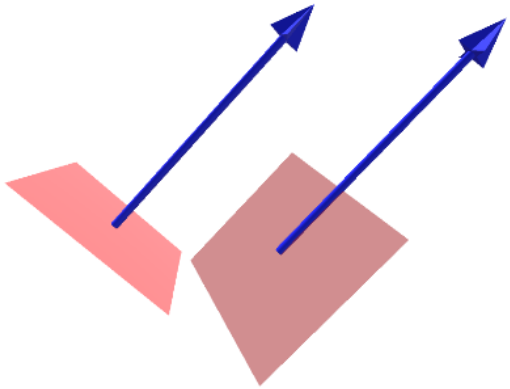
  Null

  Implementación:

  ```
  Color color = parallelogram.GetColor();
  ```

9. Graficar plano referente a un vector:

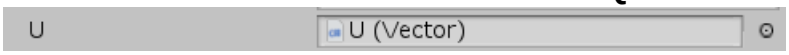
R3+ permite graficar el plano paralelo y el plano perpendicular a un vector. Este plano es dinámico, personalizable, y se actualiza automáticamente en función del vector que lo genera.



9.1. Agregar un plano referente a un vector a través de inspector:

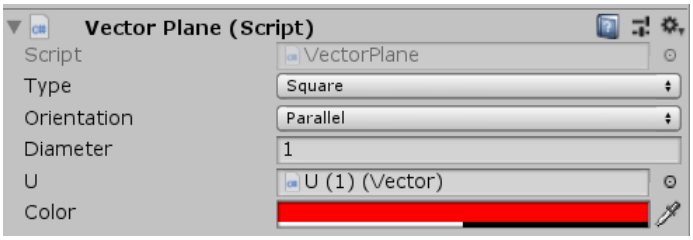
I. Crear un nuevo *GameObject*
II. Agregar el componente "*Vector Plane*"

Para que el plano referente a un vector sea visible hay que agregar el vector que lo genera a través del inspector; esto se hace en el campo U. Una vez agregado el plano se graficará y actualizará automáticamente cuando la coordenada **Q** del vector generador se modifique.

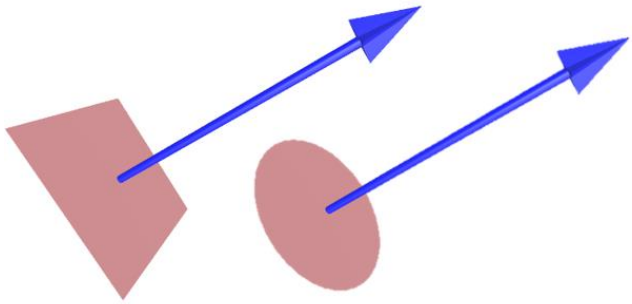


9.2. Modificar plano referente a un vector a través de inspector:

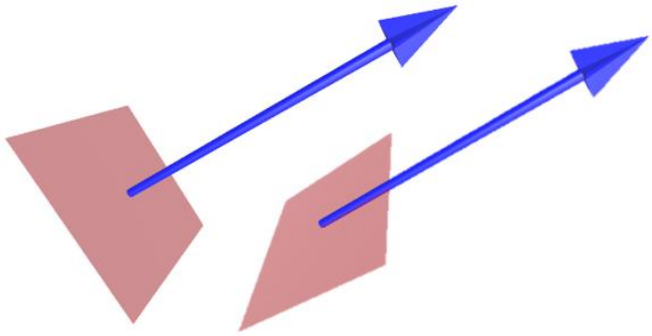
La forma en que se grafica el plano es personalizable en términos de tipo, orientación, diámetro y color.



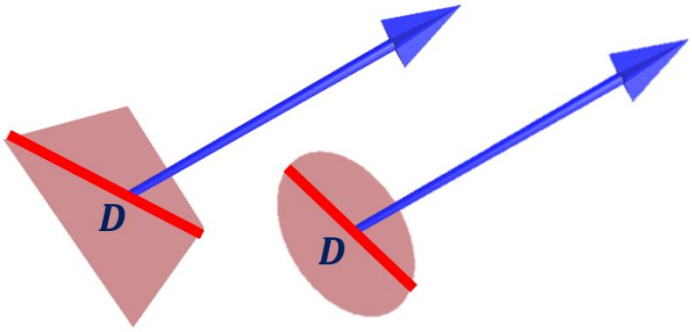
9.2.1. Type: Define la forma del plano.



9.2.2. Orientation: Permite definir la orientación del plano.



9.2.3. Diameter: Define el diámetro que tendrán los planos. En caso de ser *Circle* será el diámetro de la circunferencia, en caso de ser *Square* será la base del cuadrado.



9.2.4. Color: Define el color del ángulo

9.3. Agregar paralelogramo a través de código:

```
VectorPlane vectorPlane = VectorPlane.Create(U);
```

9.4. Métodos públicos:

- **SetVector:** define el vector referente al plano.

  Retorna:

  `Null`

  Parámetros:

  `Vector` `U`

  `Vector` `V`

Implementación:

```
SetVector(U);
```

- **SetType:** Define el tipo de plano que se graficará.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  VectorPlane.Type type
  ```

  Implementación:

  ```
  SetType(type);
  ```

- **SetOrientation:** Define la orientación del plano.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Orientation orientation
  ```

  Implementación:

  ```
  SetOrientation(orientation);
  ```

- **SetDiameter:** Define el diámetro del plano graficado. En caso de ser un círculo, define el diámetro de este, en caso de ser un cuadrado define la base de este.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float diameter
  ```

  Implementación:

  ```
  SetDiameter(diameter);
  ```

- **SetColor:** Define el color del plano.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Color color;
  ```

  Implementación:

  ```
  SetColor(color);
  ```

- **GetType:** Retorna el tipo de plano.

  Retorna:

  ```
  VectorPlane.Type
  ```

Parámetros:

null

Implementación:

```
VectorPlane.Type type = vectorPlane.GetType();
```

- **GetOrientation:** Retorna la orientación del plano.

  Retorna:

  VectorPlane.Orientation

  Parámetros:

  Null

  Implementación:

  ```
  VectorPlane.Orientation o = vectorPlane.GetOrientation ();
  ```

- **GetDiameter:** Retorna el diámetro del plano.

  Retorna:

  float

  Parámetros:

  Null

  Implementación:

  ```
  float diameter = vectorPlane.GetDiameter ();
  ```

- **GetColor:** Retorna el color del plano.

  Retorna:

  Color

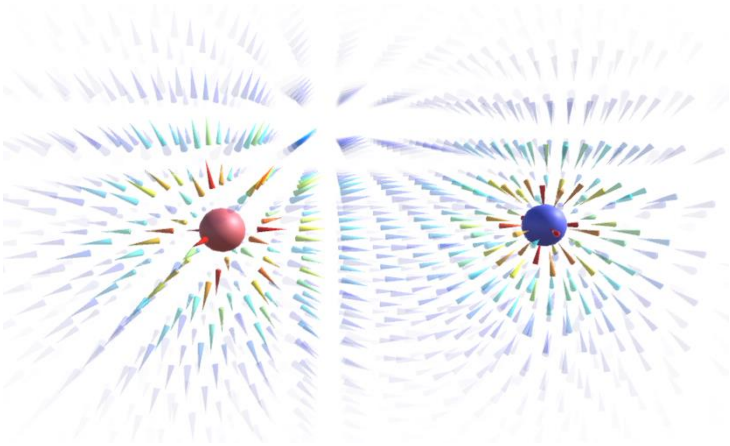
  Parámetros:

  Null

  Implementación:

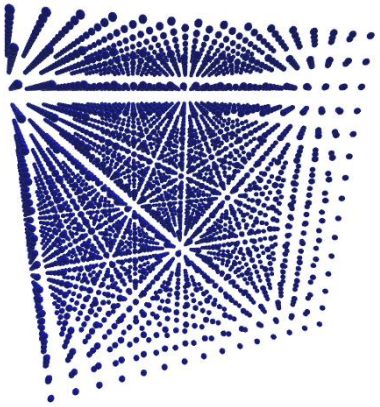
  ```
  Color color = vectorPlane.GetColor();
  ```

10. Graficar campos vectoriales:



R3+ permite graficar campos vectoriales de una manera muy dinámica, sencilla y personalizable tanto por código como por inspector. Un campo vectorial es un contenedor de elementos generadores de campo, y se actualiza dinámicamente al introducir nuevos elementos, o al actualizar las propiedades de cualquier elemento ya contenido en este. Los campos vectoriales son totalmente personalizables en términos de colormap, modelo de vector (cuerpo, cabeza, cola), norma de renderizado, modelo cuando la norma vale cero, límites y dimensiones.

10.1. Agregar un campo vectorial:



I. Crear un nuevo *GameObject*
II. Agregar el componente "*Vector Field*"

Una vez se ha agregado el campo, se genera un espacio tridimensional lleno con vectores cero. Para que el campo tenga vectores hay que agregar elementos generadores de campo. Estos serán tratados a profundidad en el siguiente literal.

10.2. Modificar campo vectorial a través de inspector:

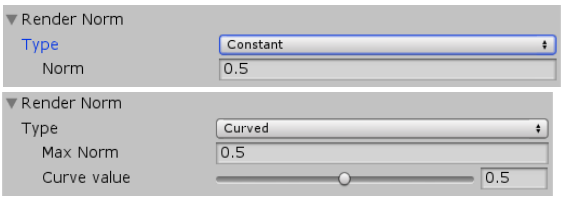
10.2.1. Field Properties: Define las propiedades de visualización del campo.

10.2.1.1. Dimension: Define la cantidad de vectores que se graficarán en cada eje.

10.2.1.2. Limits: Define desde donde hasta donde se graficarán los vectores en cada eje.

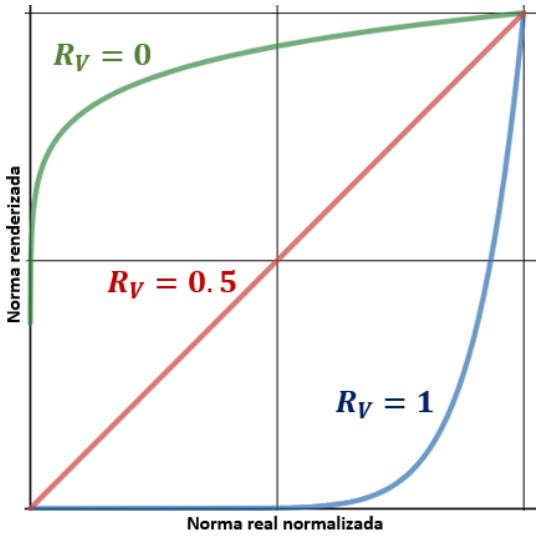
10.2.2. Vector Model: Define la forma en que se van a renderizar los vectores en la escena. Este se puede ver más a profundidad en el literal de *graficar vectores*.

10.2.3. Render Norm: Define la forma en que se va a renderizar la norma de los vectores en el campo vectorial.



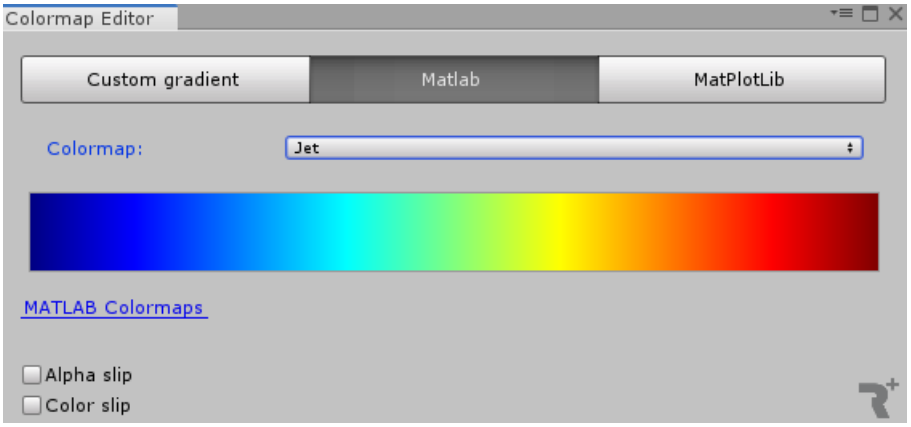
10.2.3.1. Constant: todos los vectores se grafican con la misma norma dada por el valor *Norm*.

10.2.3.2. *Curved*: los vectores se grafican con normas que van de 0 hasta *MaxNorm*, y dependiendo del parámetro *CurveValue* se definirá el tipo de ajuste que tienen las normas. Cuando *CurveValue* vale 0, el ajuste se enfoca a las normas de menor valor, cuando vale 0.5 es un ajuste perfectamente lineal y cuando es 1 es un ajuste enfocado a las normas de mayor valor.

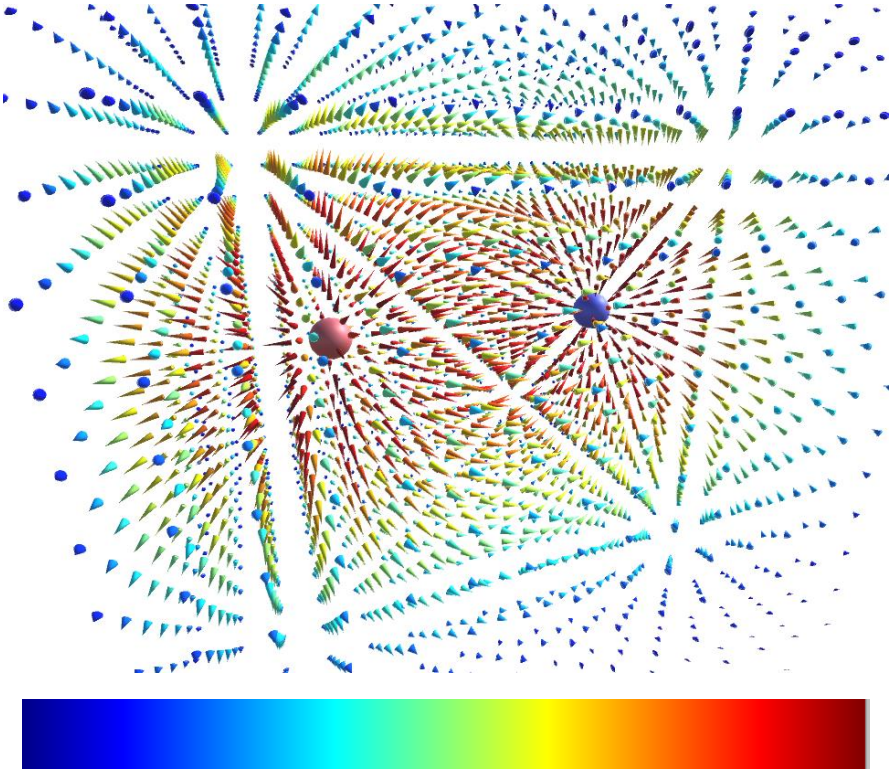


10.2.4. Material: Define el material de todos los vectores del campo.

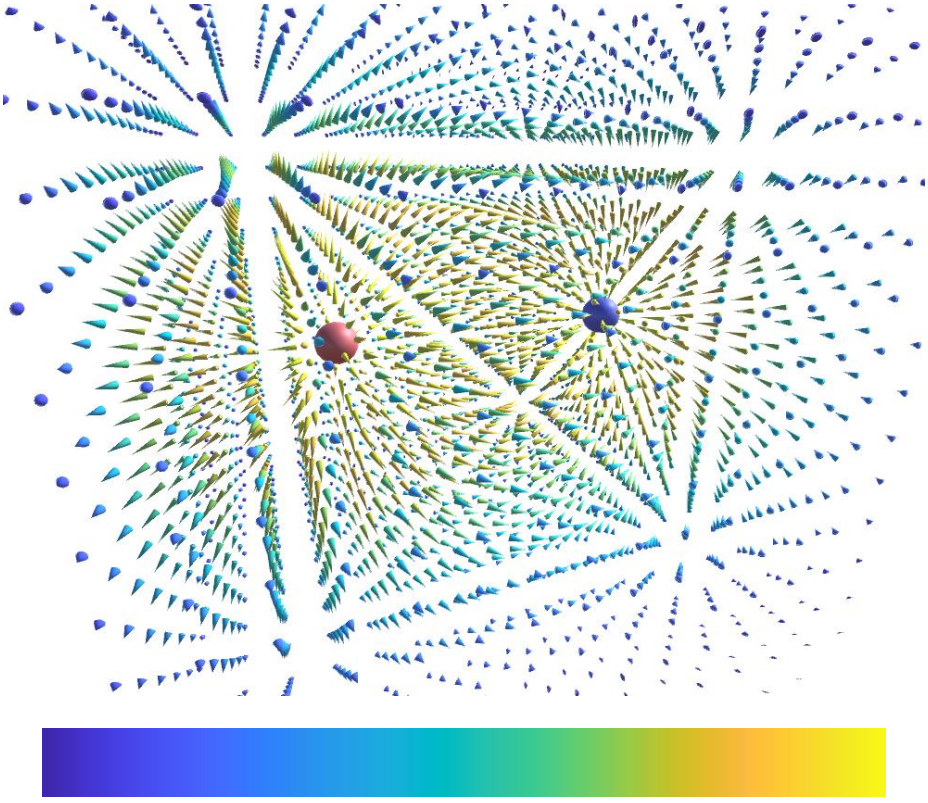
10.2.5. Colormap: Define el mapa de colores del campo. Este mapa de colores puede ser uno predeterminado de Matlab o MatPlotLib, o un gradiente de color personalizado.



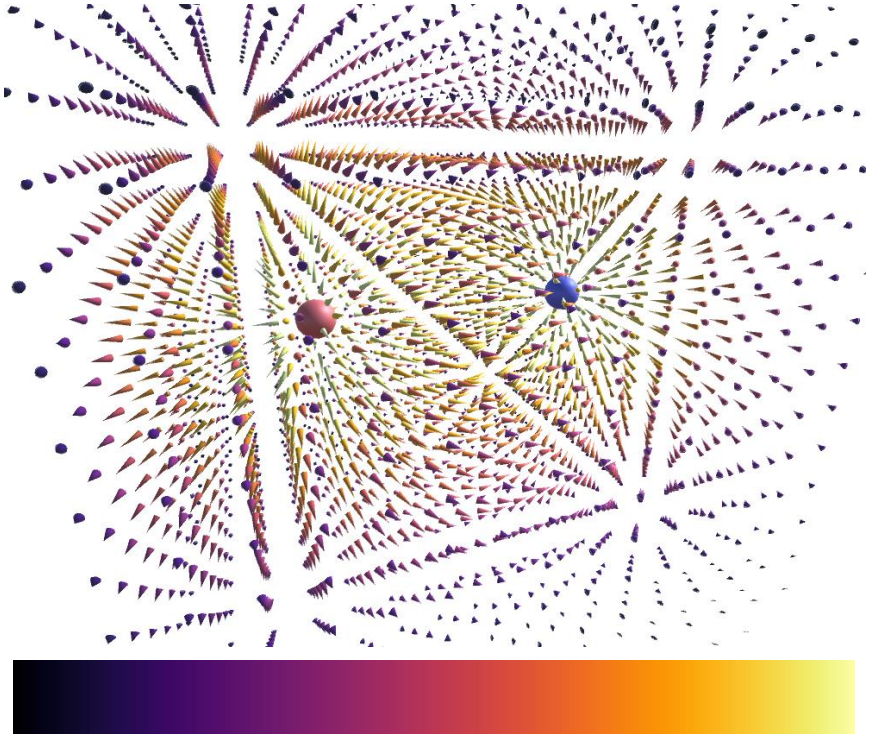
Colormap: JET



Colormap: Parula

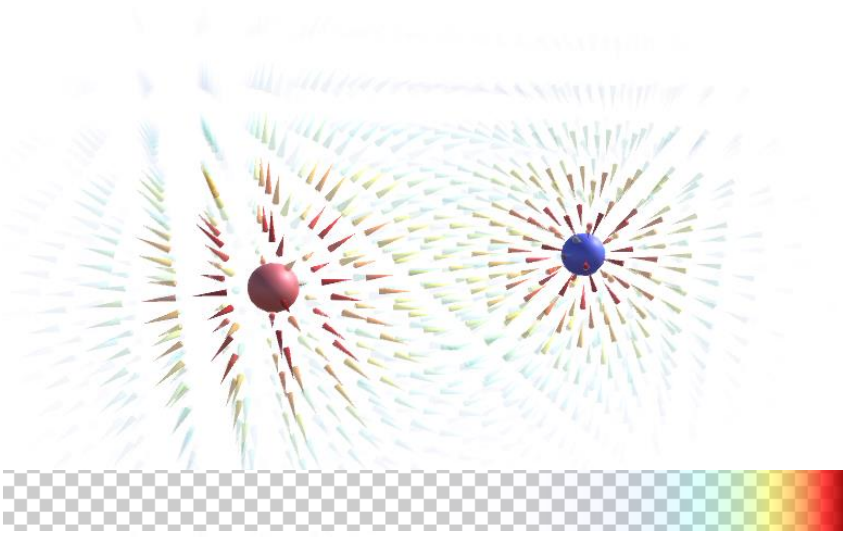


Colormap: Inferno



Se pueden hacer corrimientos de transparencia y de color para mejorar la visibilidad de los vectores que tienen una mayor norma.

Colormap: Jet con AlphaSlip = 0.9 y ColorSlip = 0.8



10.3. Agregar un campo vectorial a través de código:

```
VectorField vectorField = VectorField.Create();
```

10.4. Métodos públicos:

- **SetDimensions:** Define las dimensiones del campo vectorial.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  int x
  int y
  int z
  ```

  Implementación:

  ```
  SetDimensions(x,y,z);
  ```

- **SetAllLimits:** Define desde donde a donde van todos los ejes del campo vectorial.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float fromX
  float toX
  float fromY
  float toY
  float fromZ
  float toZ
  ```

  Implementación:

  ```
  SetAllLimits(fromX, toX, fromY, toY, fromZ, toZ);
  ```

- **SetLimits:** Define desde donde a donde va un eje en específico del campo vectorial.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Axes.Axis axis
  float from
  float to
  ```

  Implementación:

  ```
  SetLimits(axis, from, to);
  ```

- **Set1DimYZ:** Prepara el campo para solo graficar en el plano YZ.

  Retorna:
  ```
  Null
  ```
  Parámetros:
  ```
  float center
  int dimY
  int dimZ
  ```
  Implementación:
  ```
  Set1DimYZ(center, dimY, dimZ);
  ```

- **Set1DimXZ:** Prepara el campo para solo graficar en el plano XZ.

  Retorna:
  ```
  Null
  ```
  Parámetros:
  ```
  float center
  int dimX
  int dimZ
  ```
  Implementación:
  ```
  Set1DimXZ(center, dimX, dimZ);
  ```

- **Set1DimXY:** Prepara el campo para solo graficar en el plano XY.

  Retorna:
  ```
  Null
  ```
  Parámetros:
  ```
  float center
  int dimX
  int dimY
  ```
  Implementación:
  ```
  Set1DimXY(center, dimX, dimY);
  ```

- **SetHighPerformanceMode:** Prepara el campo para ser lo mas óptimo en rendimiento gráfico.

  Retorna:
  ```
  Null
  ```
  Parámetros:
  ```
  Null
  ```
  Implementación:
  ```
  SetHighPerformanceMode();
  ```

- **SetAsEquipotentialSurface:** Prepara el campo para graficar una superficie equipotencial.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Radius (Opcional)
  ```

  Implementación:

  ```
  SetAsEquipotentialSurface(Radius)
  ```

- **SetVectorModelType:** Define el tipo de modelo que utilizarán todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  VectorModel.Type type
  ```

  Implementación:

  ```
  SetVectorModelType(type);
  ```

- **SetConeDivs:** Define cuantas divisiones va a tener la base del cono (En caso se ser el modelo de cuerpo seleccionado) de todos los vectores del campo**.**

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  int divs
  ```

  Implementación:

  ```
  SetConeDivs(divs);
  ```

- **SetHeadModel:** Define el modelo con el que se renderiza la cabeza de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetHeadModel(model);
  ```

- **SetHeadRadius:** Define el radio con el que se renderiza la cabeza de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetHeadRadius(radius);
  ```

- **SetHeadLong:** Define el largo con el que se renderiza la cabeza de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float Long
  ```

  Implementación:

  ```
  SetHeadLong(Long);
  ```

- **SetBodyModel:** Define el modelo con el que se renderiza el cuerpo de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetBodyModel(model);
  ```

- **SetBodyRadius:** Define el radio con el que se renderiza el cuerpo de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetBodyRadius(radius);
  ```

- **SetTailModel:** Define el modelo con el que se renderiza la cola de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

Parámetros:

```
GameObject model
```

Implementación:

```
SetTailModel(model);
```

- **SetTailRadius:** Define el radio con el que se renderiza la cola de todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

  Implementación:

  ```
  SetTailRadius(radius);
  ```

- **SetZeroModel:** Define el modelo con el que se renderiza un vector cuando su norma vale cero.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  GameObject model
  ```

  Implementación:

  ```
  SetTailModel(model);
  ```

- **SetZeroRadius:** Define el radio con el que se renderiza un vector cuando su norma vale cero.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float radius
  ```

- **SetRenderNormType:** Define la forma en que se renderiza la norma del vector.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  VectorProps.RenderNormVF.Type type
  ```

  Implementación:

  ```
  SetRenderNormType(type);
  ```

- **SetRenderNormConstant:** Define el largo con el que se renderiza el vector si el type es *Constant*. En caso de ser *Curved,* el *type* es cambiado automáticamente a *Constant.*

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float norm;
  ```

  Implementación:

  ```
  SetRenderNormConstant(norm);
  ```

- **SetMaxNormValue:** Define el valor de norma máximo con el que se renderizan los vectores.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float maxNorm
  ```

  Implementación:

  ```
  SetMaxNormValue(maxNorm);
  ```

- **SetCurveValue:** Define el valor de ajuste curvo de las normas de los vectores. Este parámetro varía entre 0 y 1.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  float curve
  ```

  Implementación:

  ```
  SetCurveValue(curve);
  ```

- **SetMaterial:** Define el material con el que se renderizan todos los vectores del campo.

  Retorna:

  ```
  Null
  ```

  Parámetros:

  ```
  Material material
  ```

  Implementación:

  ```
  SetMaterial(material);
  ```

- **SetColormap:** Define el colormap con el que se renderizan todos los vectores del campo.

  Retorna:
  Null
  Parámetros:
  Colormap colormap
  Implementación:
  ```
  SetColormap(colormap);
  ```

- **GetDimension:** Retorna la dimensión del campo vectorial.

  Retorna:
  Vector3
  Parámetros:
  Null
  Implementación:
  ```
  Vector3 dimension = vectorField.GetDimension();
  ```

- **GetLimits:** Retorna los límites de graficación del campo de un eje en particular.

  Retorna:
  Vector2
  Parámetros:
  Axes.axis axis
  Implementación:
  ```
  Vector2 limits = vectorField.GetLimits(axis);
  ```

- **GetModelType:** Retorna el tipo de modelo que usan todos los vectores del campo.

  Retorna:
  VectorModel.Type
  Parámetros:
  Null
  Implementación:
  ```
  VectorModel.Type type = vectorField.GetType();
  ```

- **GetConeDivs:** Retorna el número de divisiones que tiene el modelo cono.

  Retorna:
  int
  Parámetros:
  Null
  Implementación:
  ```
  int divs = vectorField.GetConeDivs();
  ```

- **GetHeadModel:** Retorna el modelo de la cabeza de todos los vectores del campo.

  Retorna:

  ```
  GameObject
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  GameObject model = vectorField.GetHeadModel();
  ```

- **GetHeadRadius:** Retorna el radio de la cabeza de todos los vectores del campo.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float radius = vectorField.GetHeadRadius();
  ```

- **GetHeadLong:** Retorna el largo de la cabeza de todos los vectores del campo.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float Long = vectorField.GetHeadLong();
  ```

- **GetBodyModel:** Retorna el modelo del cuerpo de todos los vectores del campo.

  Retorna:

  ```
  GameObject
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  GameObject model = vectorField.GetBodyModel();
  ```

- **GetBodyRadius:** Retorna el radio del cuerpo de todos los vectores del campo.

  Retorna:

  ```
  float
  ```

  Parámetros:

  ```
  Null
  ```

  Implementación:

  ```
  float radius = vectorField.GetBodyRadius();
  ```

- **GetTailModel:** Retorna el modelo de la cola de todos los vectores del campo.

  Retorna:

  `GameObject`

  Parámetros:

  `Null`

  Implementación:

  ```
  GameObject model = vectorField.GetTailModel();
  ```

- **GetTailRadius:** Retorna el radio del modelo de la cola de todos los vectores del campo.

  Retorna:

  `float`

  Parámetros:

  `Null`

  Implementación:

  ```
  float radius = vectorField.GetTailRadius();
  ```

- **GetZeroModel:** Define el modelo con el que se renderiza un vector cuando su norma vale cero.

  Retorna:

  `GameObject`

  Parámetros:

  `Null`

  Implementación:

  ```
  GameObject model = vectorField.GetZeroModel();
  ```

- **GetZeroRadius:** Retorna el radio con el que se renderiza un vector cuando su norma vale cero.

  Retorna:

  `float`

  Parámetros:

  `Null`

  Implementación:

  ```
  float radius = vectorField.GetZeroRadius();
  ```

- **GetRenderNormType:** Retorna el tipo de norma con la que se están renderizando los vectores en el campo.

  Retorna:

  `VectorProps.RenderNormVF.Type`

Parámetros:

Null

Implementación:

```
VectorProps.RenderNormVF.Type t = vectorField.GetRenderNormType();
```

- **GetRenderNormValue:** En caso de ser *constant,* Retorna el valor de la norma con la que se están renderizando todos los vectores del campo. En caso de ser *curve* retorna el valor de curve.

  Retorna:

  `float`

  Parámetros:

  Null

  Implementación:

  ```
  float norm = vectorField.GetRenderNormValue();
  ```

- **GetMaterial:** Retorna el material con el que se renderizan todos los vectores del campo.

  Retorna:

  `Material`

  Parámetros:

  Null

  Implementación:

  ```
  Material material = vectorField.GetMaterial();
  ```

- **GetColorMap:** Retorna el Colormap con el que se están graficando los vectores del campo.

  Retorna:

  `Colormap`

  Parámetros:

  Null

  Implementación:

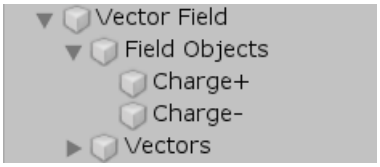
  ```
  Colormap colormap = vectorField.GetColorMap();
  ```

11. Objetos generadores de campo:

Son los encargados de modelar la forma del campo a través de una función matemática que va de $\mathbb{R}^3$ a $\mathbb{R}^3$ en coordenadas cartesianas, de la forma:

$$E_{objeto} = f(x,y,z) = \begin{bmatrix} f_x(x,y,z) \\ f_y(x,y,z) \\ f_z(x,y,z) \end{bmatrix}$$

11.1.1. Agregar un objeto generador de campo a un campo vectorial:



I. Crear un nuevo *GameObject*
II. Agregar el cualquier componente que herede de la clase "*FieldObject*"
III. Poner el *GameObject* creado dentro del objeto *FieldObjects,* que es hijo del objeto *VectorField*

Al agregar un objeto generador de campo al campo vectorial, automáticamente se actualizan todos los vectores. El campo puede contener varios elementos generadores de campo, y la gráfica es la sumatoria de los campos individuales de cada objeto generador de campo.

$$E_{graficado} = \sum E_{objeto}$$

11.1.2. Crear un objeto generador de campo:

Para crear un objeto generador de campo hay que crear un nuevo script, y hacer que herede de la clase *FieldObject*. Al hacer esto el compilador pedirá que se sobrescriban los siguientes métodos abstractos:

- **OnCreate:** Función que se ejecuta al crear el objeto generador de campo. Esta función puede venir vacía si no hay nada que se necesite ejecutar al crear el objeto.

- **Draw**: Función que contiene todo lo adicional que se requiera graficar y actualizar (como por ejemplo la esfera que representa gráficamente una carga eléctrica). Esta función puede ir vacía si no se requiere graficar nada adicional.

- **Function**: Aquí se define la función vectorial que va a seguir el objeto generador de campo. Toma un Vector3 como parámetro y lo procesa a través de la ecuación vectorial que el usuario defina, para finalmente retornar otro Vector3.

En esencia solo esos 3 métodos son necesarios para crear un objeto generador de campo, pero cualquier desarrollador puede complementar su clase agregando todo lo que considere necesario para tener un objeto generador de campo más completo.

La biblioteca incluye dos ejemplos de objetos generadores de campo: *ElectricCharge* que es un ejemplo bastante completo, y *CustomFunction*, que es un ejemplo básico, pero tiene todo lo necesario para ser funcional.

11.2. Agregar un objeto generador de campo a un vectorial a través de código:

```
vectorField.AddFieldObject(objetoGeneradorDeCampo);
```

12. Clases propias de la biblioteca:

12.1. Axes.Direction: Sirve para definir la dirección a la que apunta un eje.

- Negative
- Positive
- Real

12.2. Axes.Axis: Sirve como parámetro de entrada para las funciones que requieren utilizar un eje en particular.

- X
- Y
- Z

12.3. Axes.Plane: Sirve como parámetro de entrada para las funciones que requieren utilizar un plano en particular.

- XY
- XZ
- YZ

12.4. Axes.Gridlines: Sirve como parámetro de entrada para las funciones que requieren utilizar una línea de grilla en particular, o un conjunto de líneas de grillas que están en un plano en particular.

- PlaneXY
- PlaneXZ
- PlaneYZ
- XY
- XZ
- YX
- YZ
- ZX
- ZY

12.5. MathV.Coordinates: Sirve como parámetro de entrada para las funciones que requieren utilizar un sistema coordenado en particular.

- Cartesian
- Cylindrical
- Spherical

12.6. VectorModel.Type: Sirve como parámetro de entrada para las funciones que requieren utilizar un tipo de modelo en particular para graficar un vector.

- BodyAndTail
- Cone
- EquipotentialSurface

- HeadAndBody
- HeadBodyAndTail
- SingleBody

12.7. **VectorProps.RenderNormSV.Type:** Sirve como parámetro de entrada para las funciones que requieren definir la forma en que renderiza la norma de un vector.
- Constant
- Real

12.8. **VectorProps.RenderNormVF.Type:** Sirve como parámetro de entrada para las funciones que requieren definir la forma en que renderiza la norma de los vectores de un campo vectorial.
- Constant
- Curve

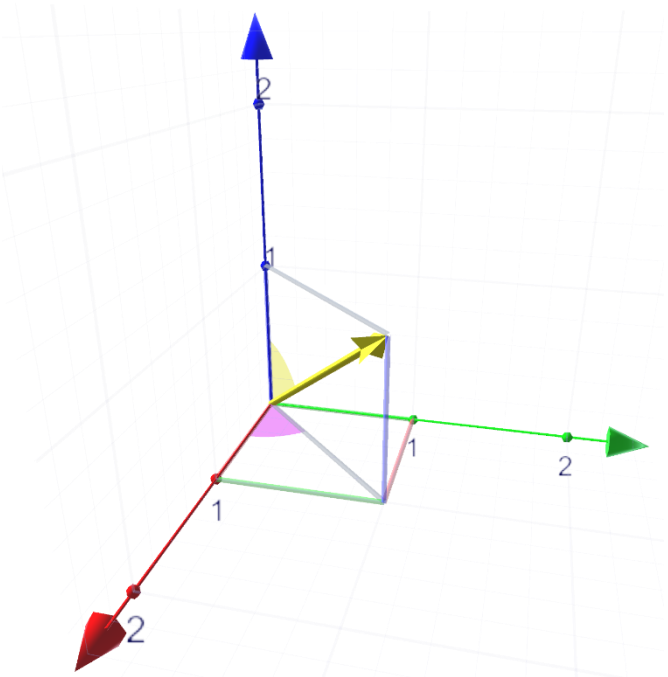
12.9. **Projections.projection:** Sirve como parámetro de entrada para las funciones que requieren definir un tipo de proyección de vector en particular.
- X0
- X1
- Y0
- Y1
- Z0
- Z1
- Ro0
- Ro1
- Theta
- Phi

12.10. **DiffVector.StartFrom:** Sirve como parámetro de entrada para las funciones que requieren definir el origen desde donde partirá el vector resta.
- Origin
- U

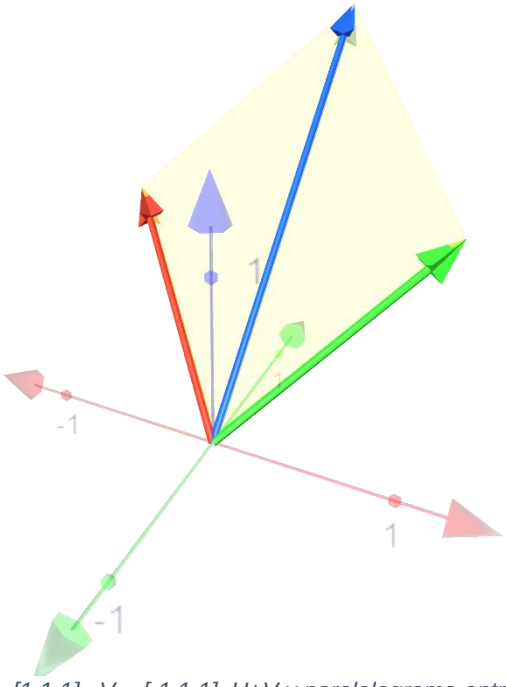
12.11. **VectorPlane.Type:** Sirve como parámetro de entrada para las funciones que requieren definir la forma que tendrá el plano referente a un vector.
- Circle
- Square

CAPITULO IV

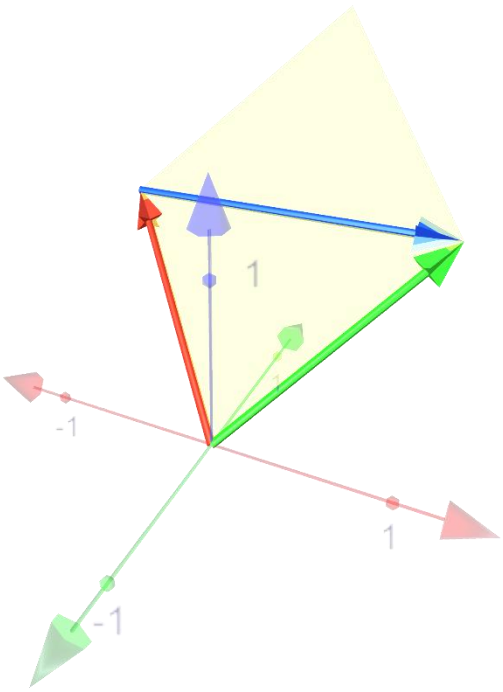
EJEMPLOS DE GRÁFICAS GENERADAS CON R3+:



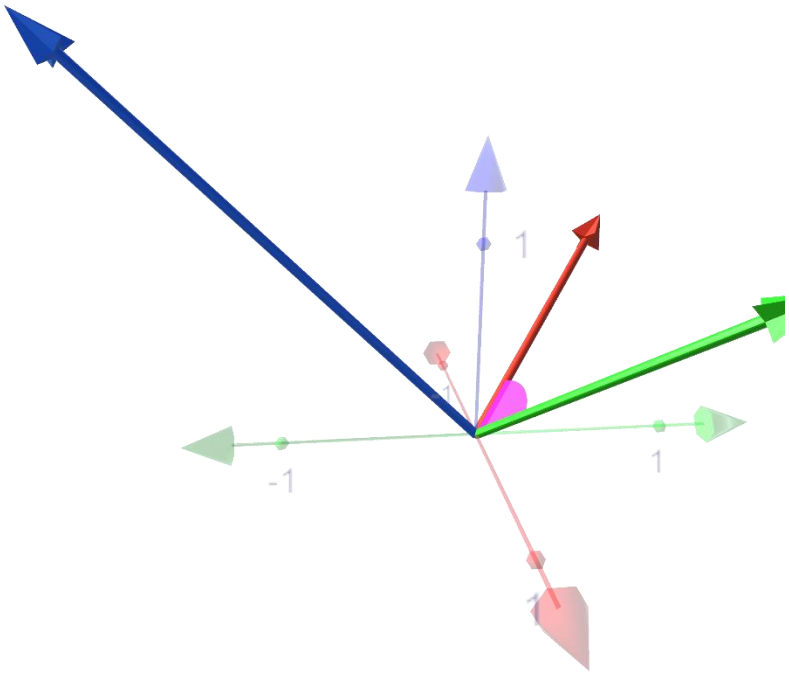
Vector U = [1 1 1] con proyecciones y eje



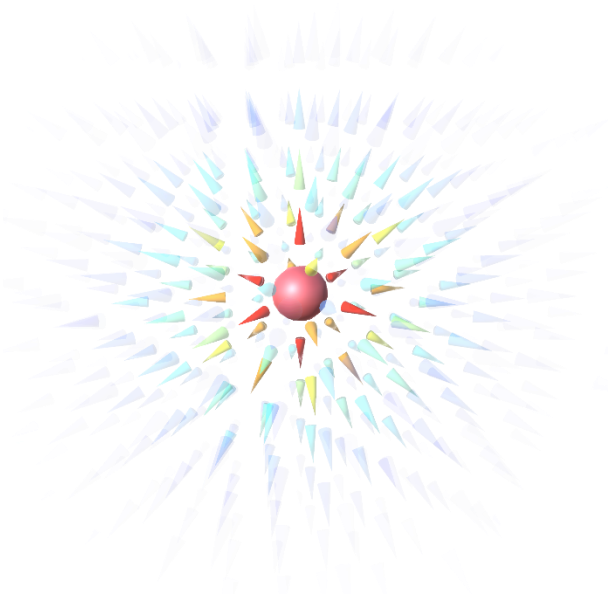
Vectores: U = [1 1 1] , V = [-1 1 1], U+V y paralelogramo entre U y V



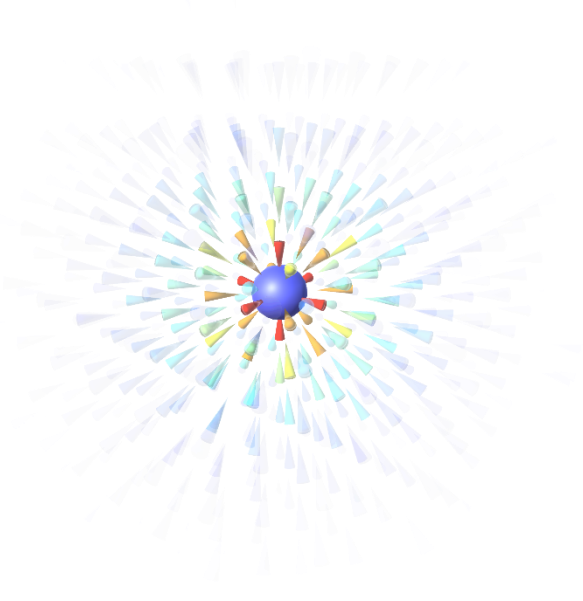
Vectores: U = [1 1 1] , V = [-1 1 1], U-V y paralelogramo entre U y V



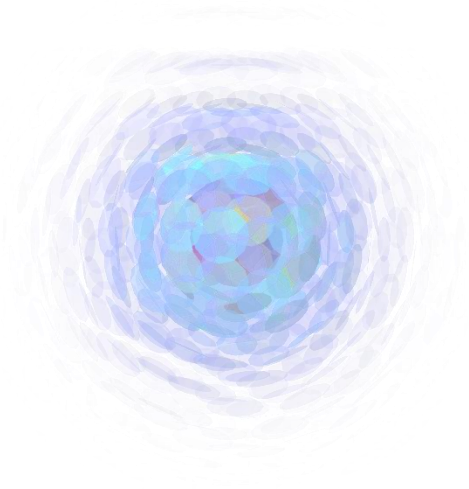
Vectores: U = [1 1 1] , V = [-1 1 1], UxV y ángulo entre U y V



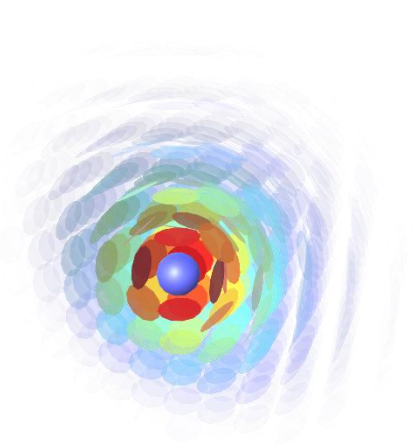
Campo vectorial: +Q en [0 0 0]



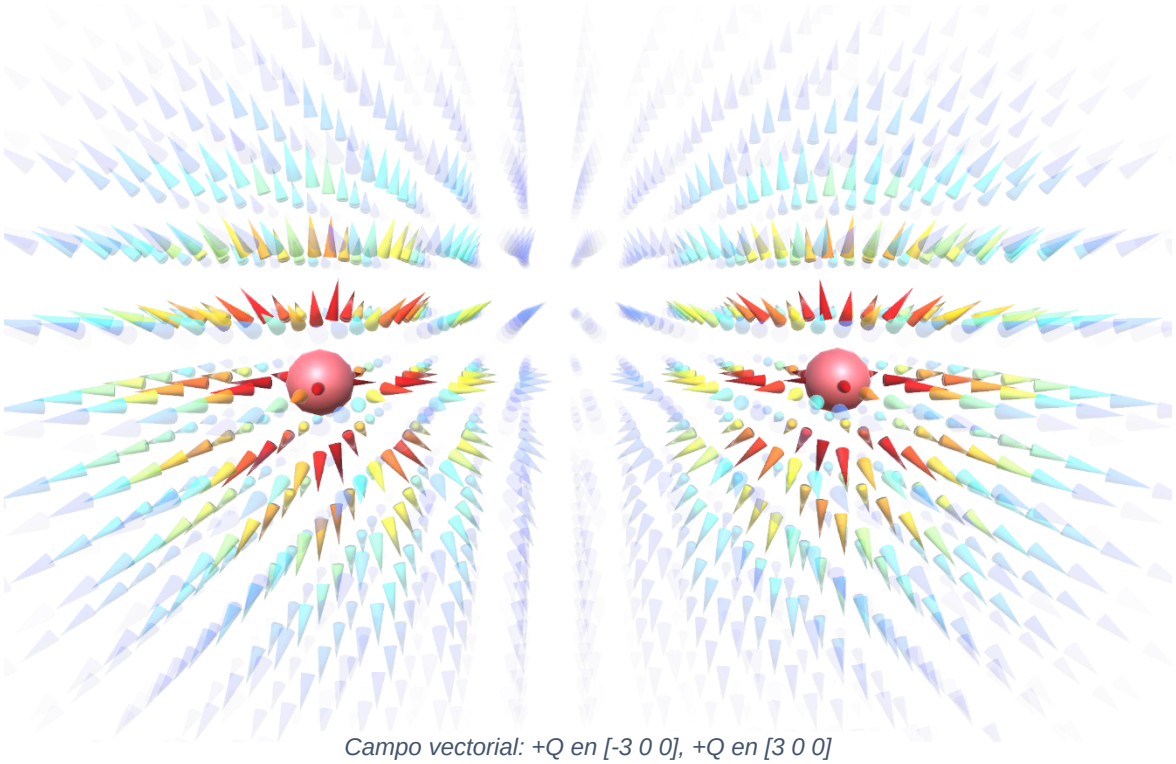
Campo vectorial: -Q en [0 0 0]



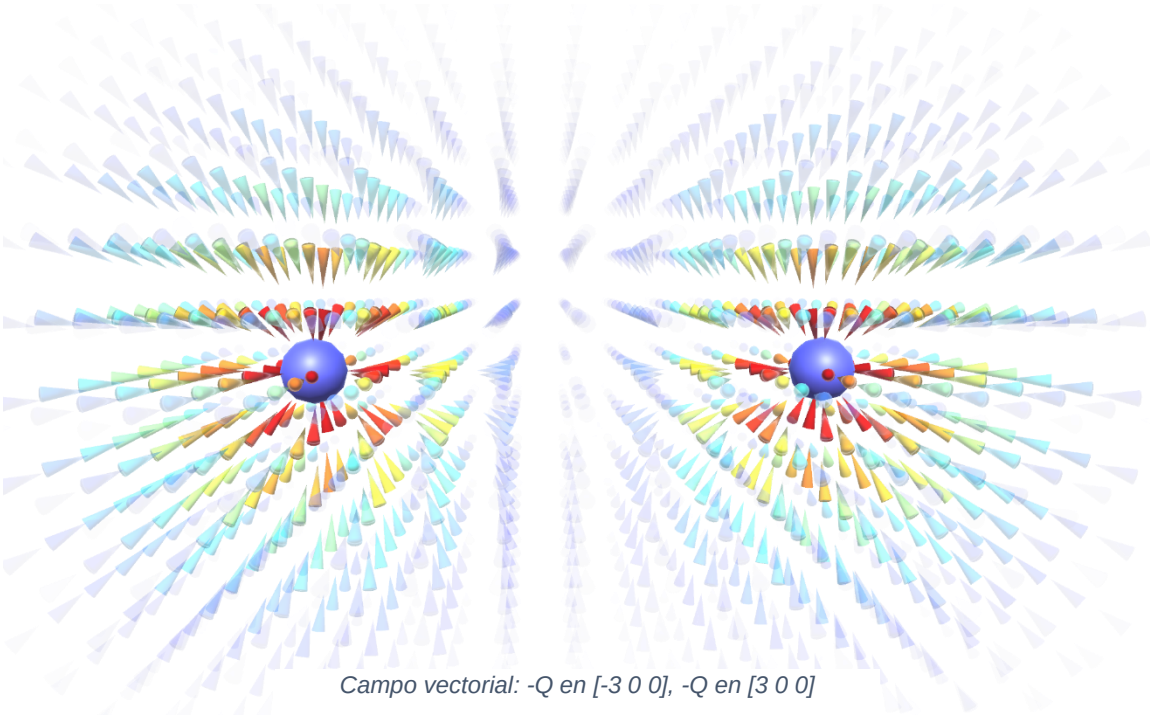
Superficie equipotencial +Q en [0 0 0]



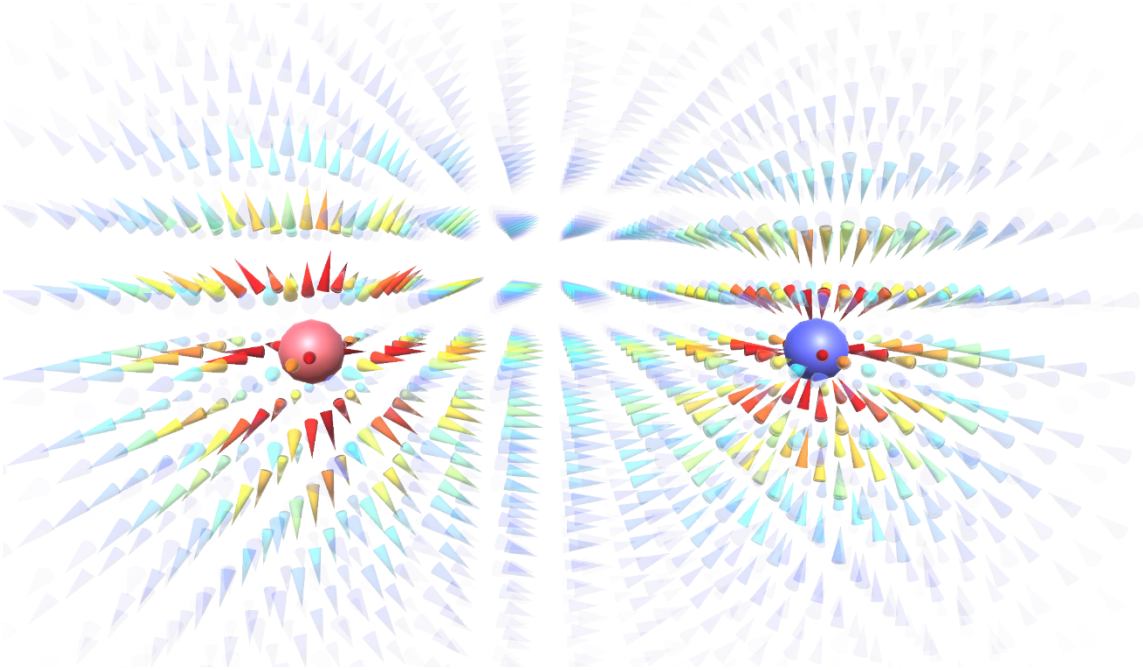
Superficie equipotencial abierta -Q en [0 0 0]



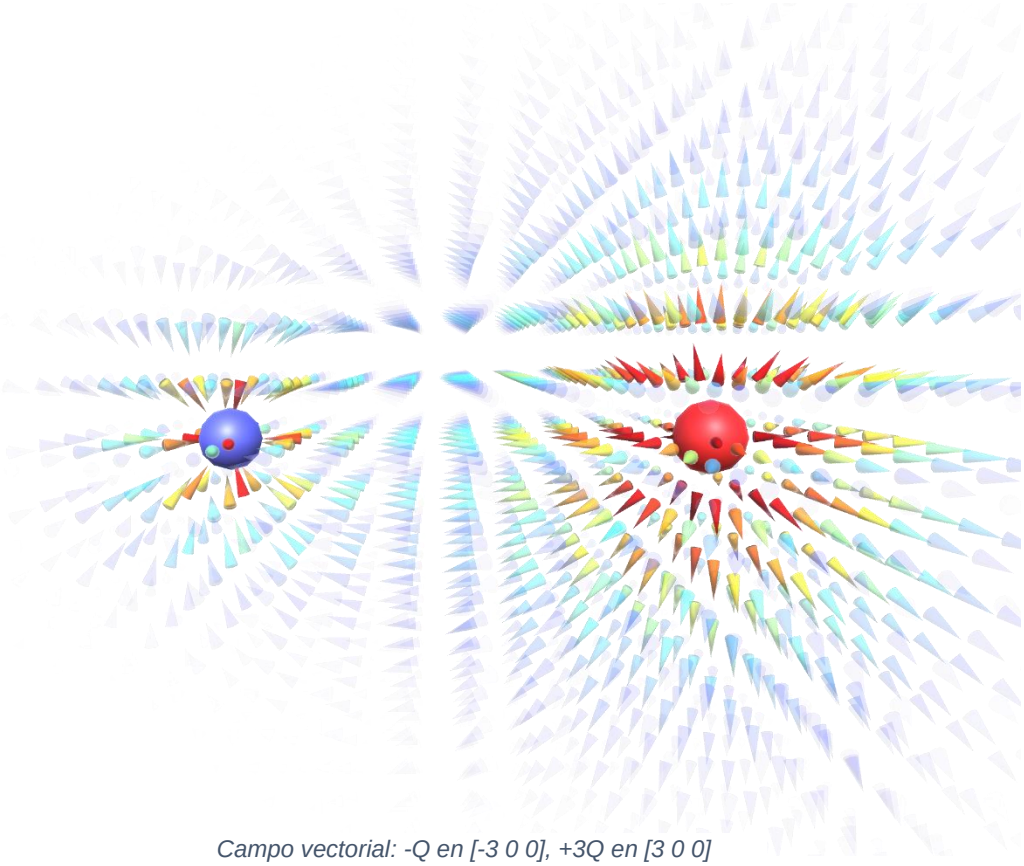
Campo vectorial: +Q en [-3 0 0], +Q en [3 0 0]



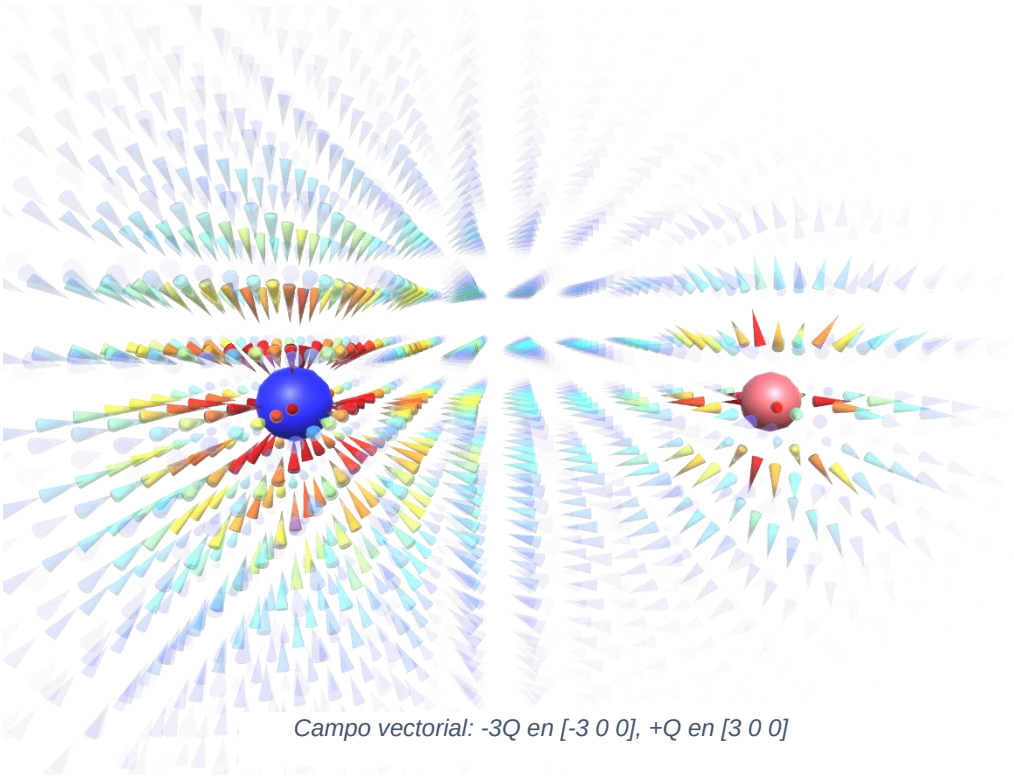
Campo vectorial: -Q en [-3 0 0], -Q en [3 0 0]



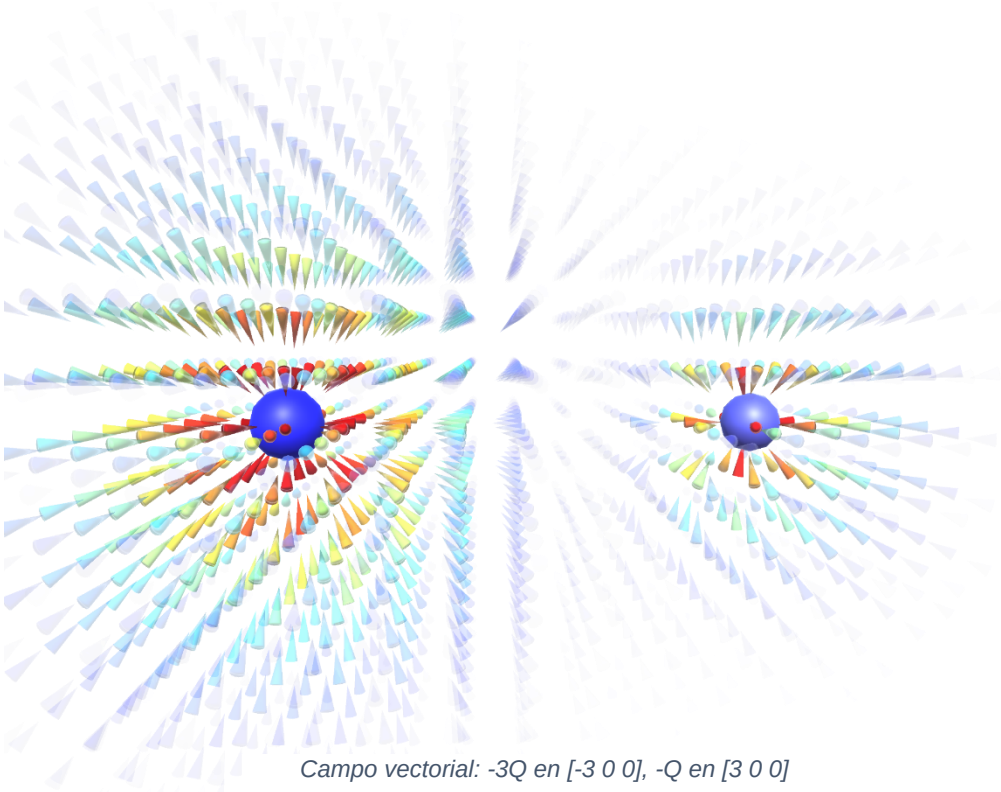
Campo vectorial: +Q en [-3 0 0], -Q en [3 0 0]



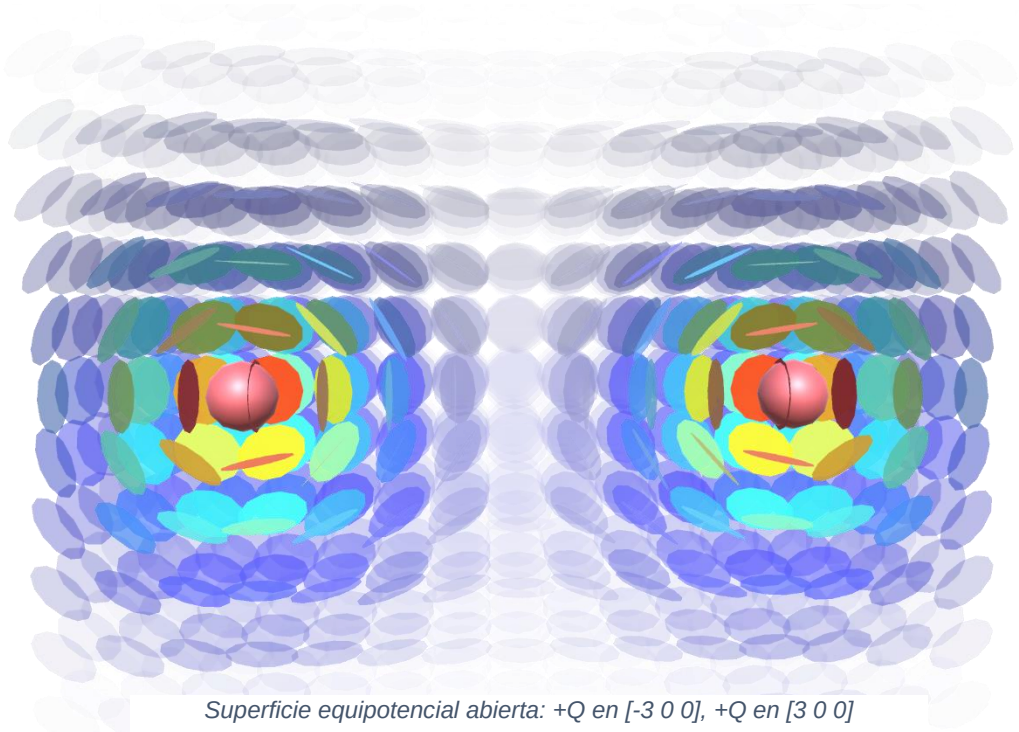
Campo vectorial: -Q en [-3 0 0], +3Q en [3 0 0]



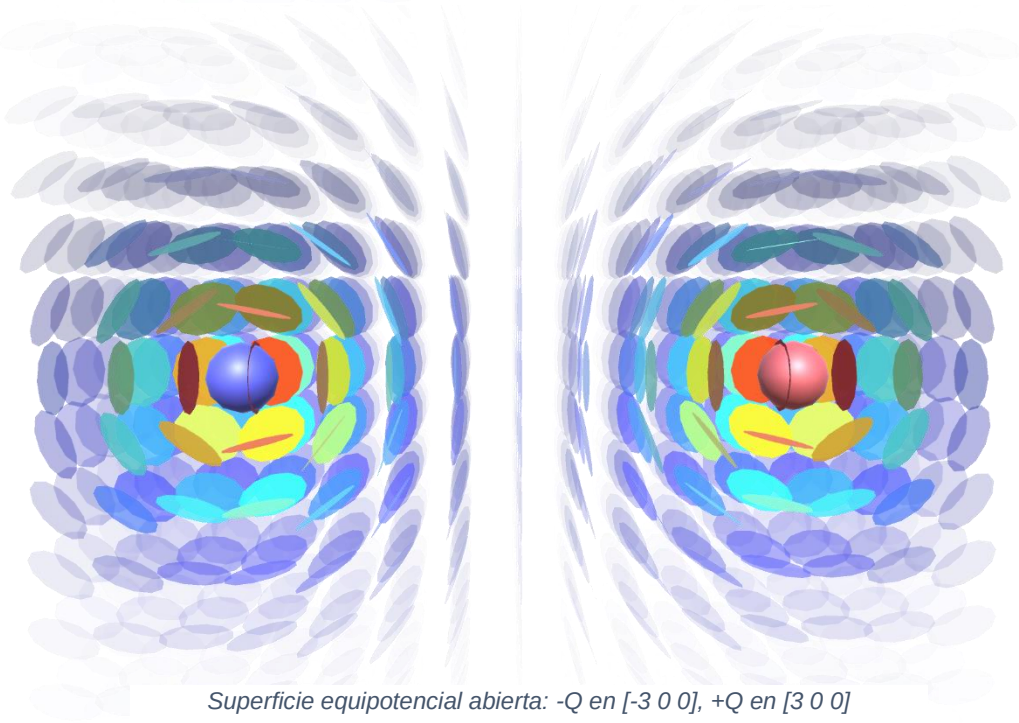
Campo vectorial: -3Q en [-3 0 0], +Q en [3 0 0]



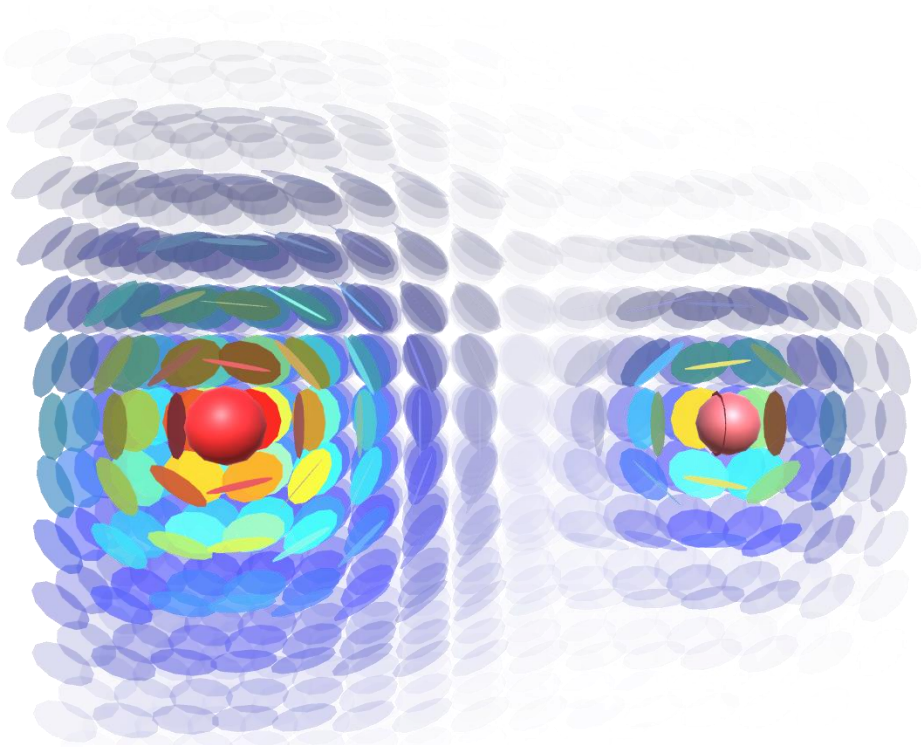
Campo vectorial: -3Q en [-3 0 0], -Q en [3 0 0]



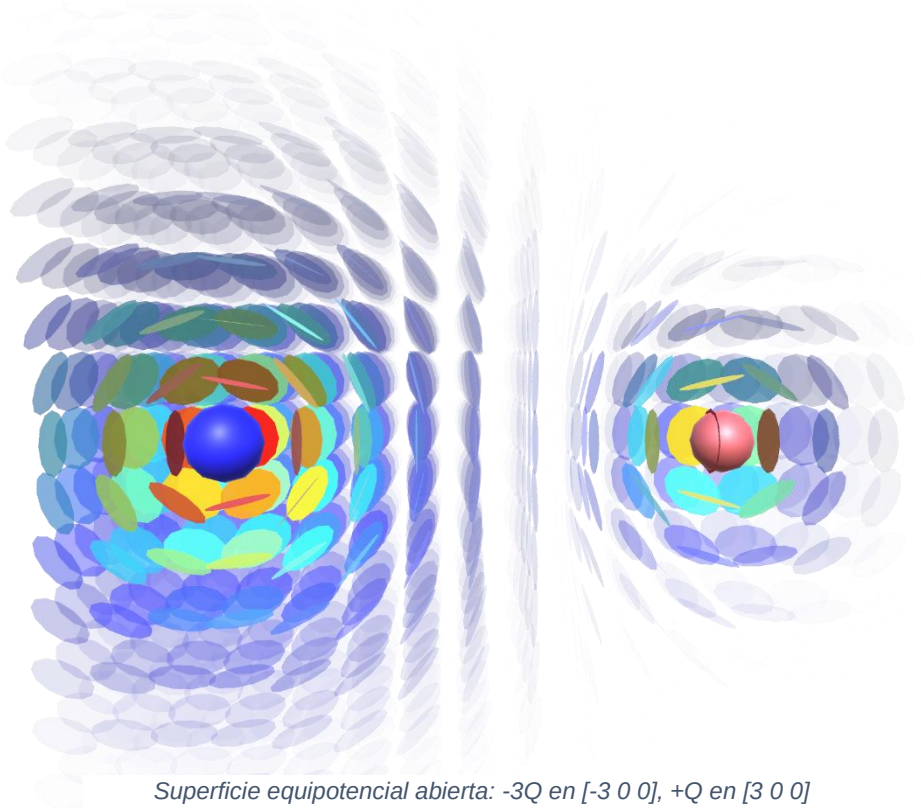
Superficie equipotencial abierta: +Q en [-3 0 0], +Q en [3 0 0]



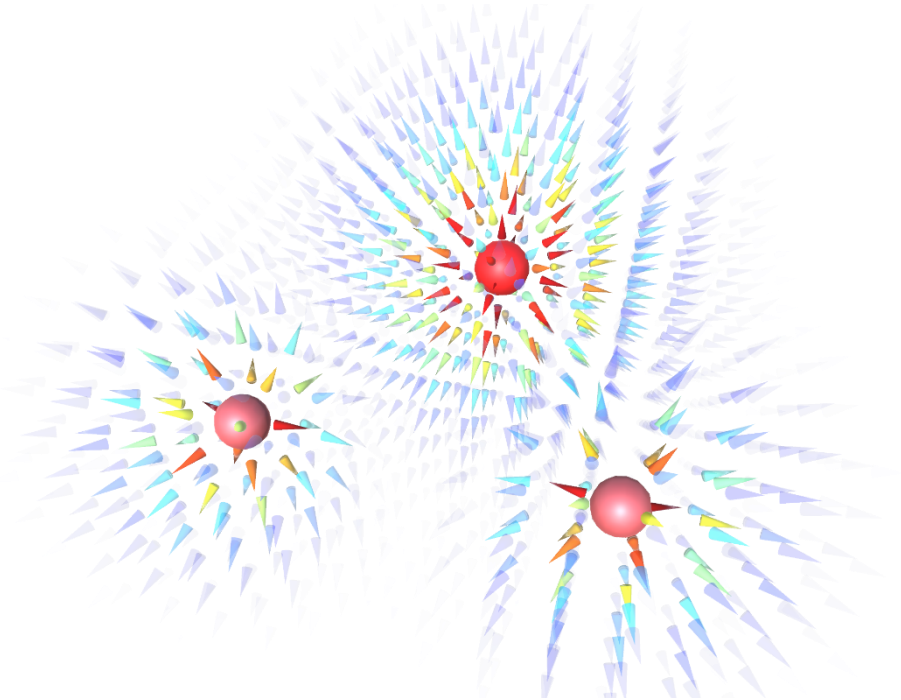
Superficie equipotencial abierta: -Q en [-3 0 0], +Q en [3 0 0]



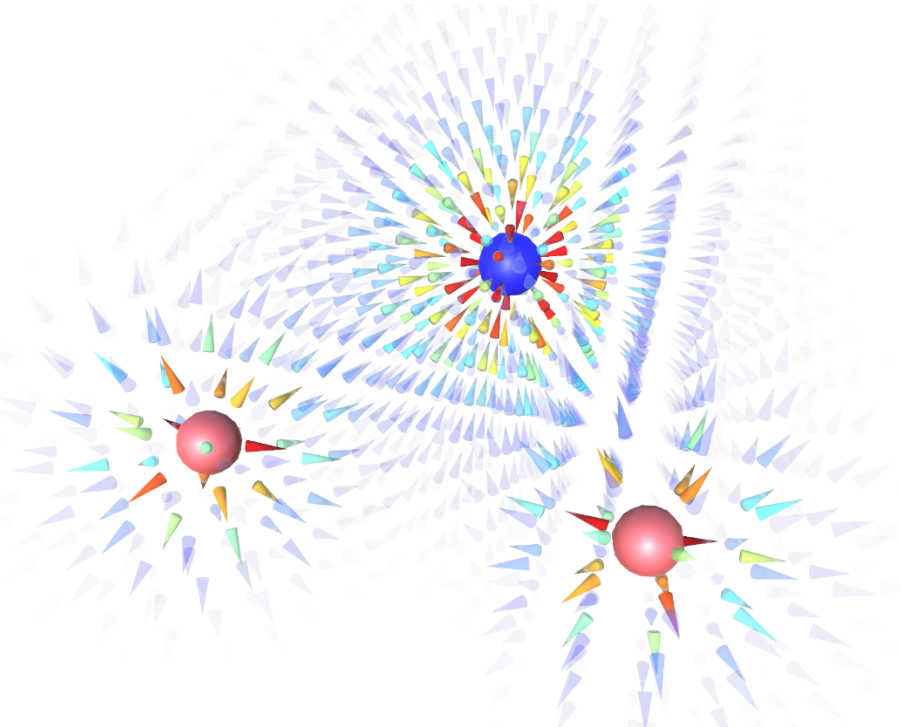
Superficie equipotencial abierta: +3Q en [-3 0 0], +Q en [3 0 0]



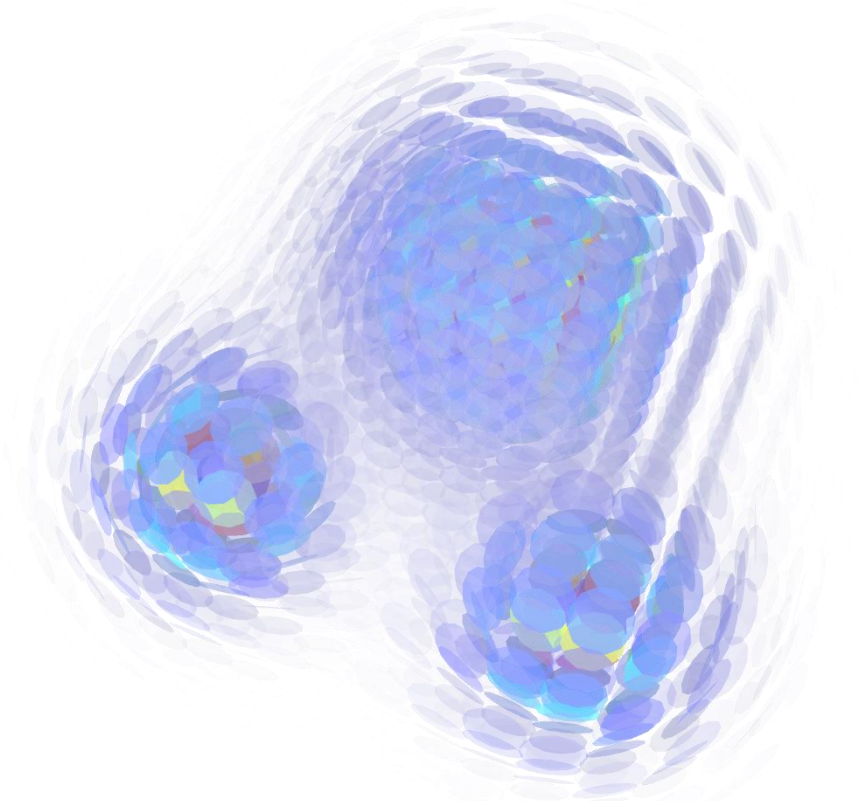
Superficie equipotencial abierta: -3Q en [-3 0 0], +Q en [3 0 0]



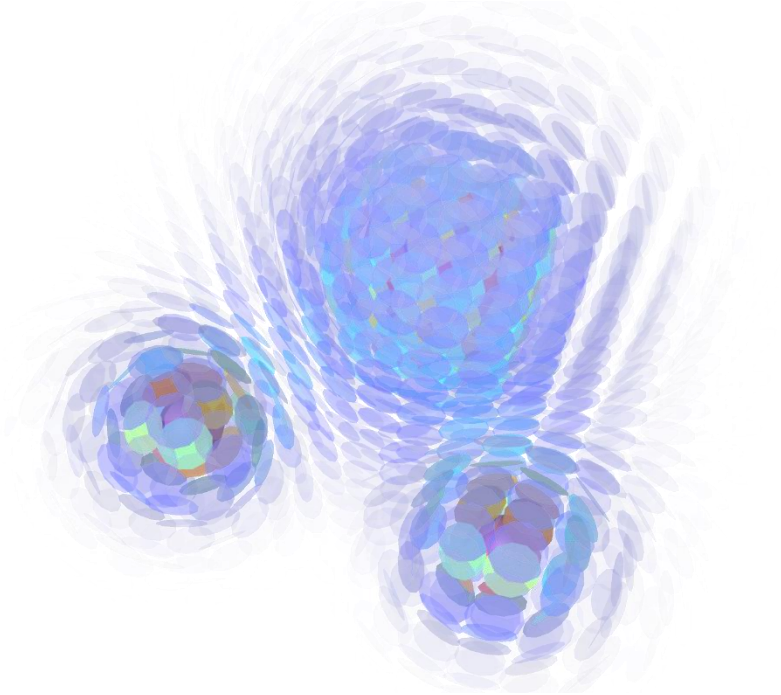
Campo vectorial: +3Q [3 0 0], +Q [-1.5 0 2.5981], +Q [-1.5 0 -2.5981]



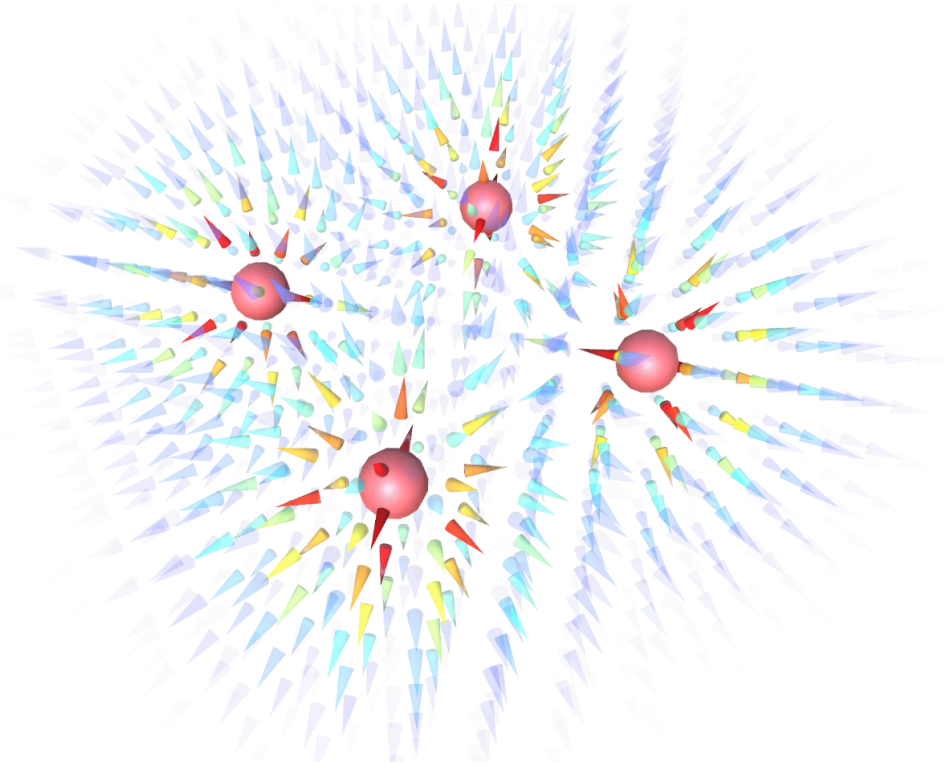
Campo vectorial: -3Q [3 0 0], +Q [-1.5 0 2.5981], +Q [-1.5 0 -2.5981]



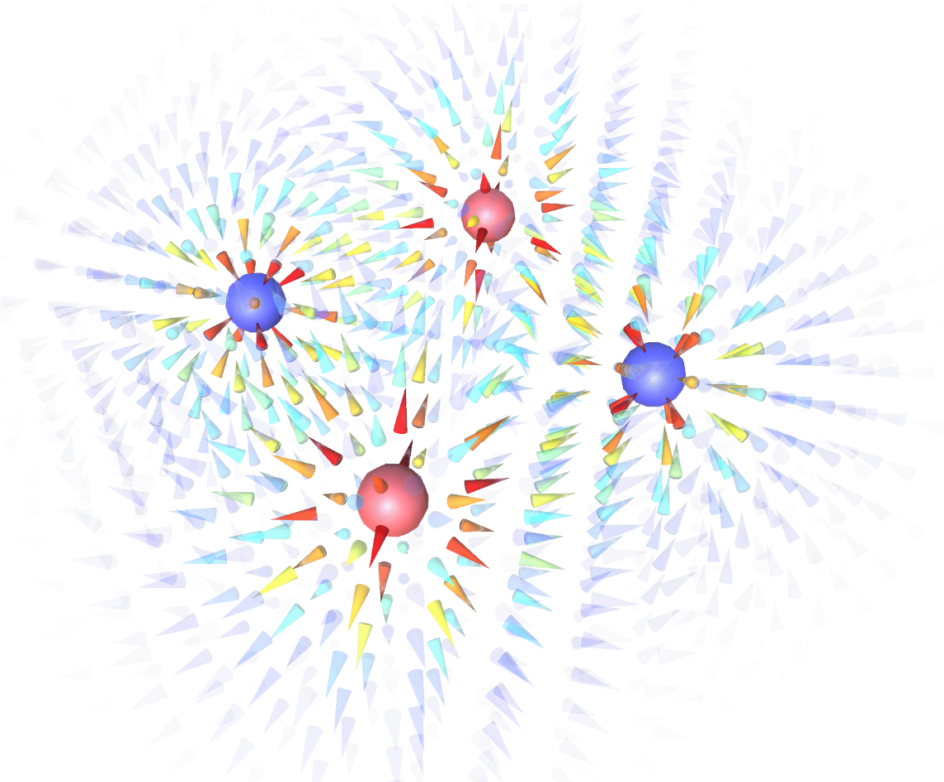
Superficie equipotencial: +3Q [3 0 0], +Q [-1.5 0 2.5981], +Q [-1.5 0 -2.5981]



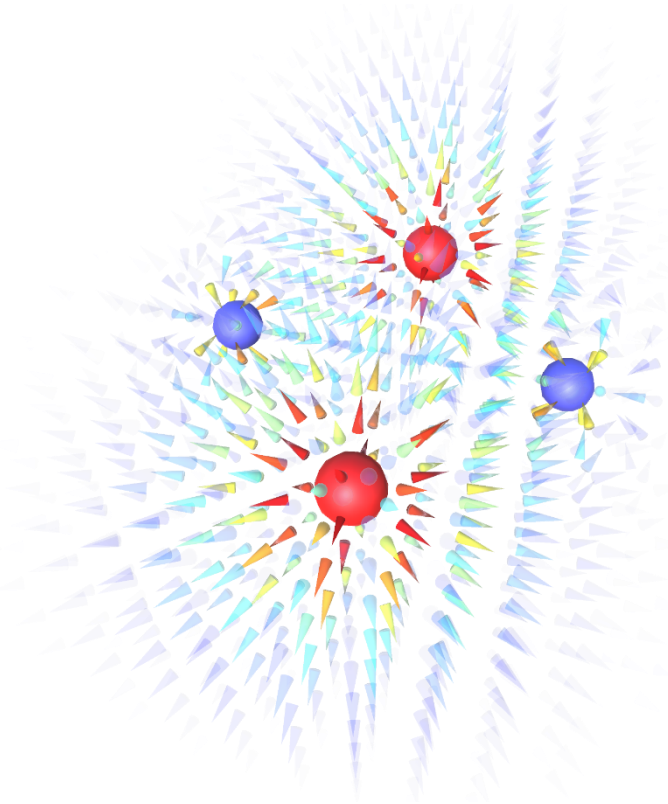
Superficie equipotencial: -3Q [3 0 0], +Q [-1.5 0 2.5981], +Q [-1.5 0 -2.5981]



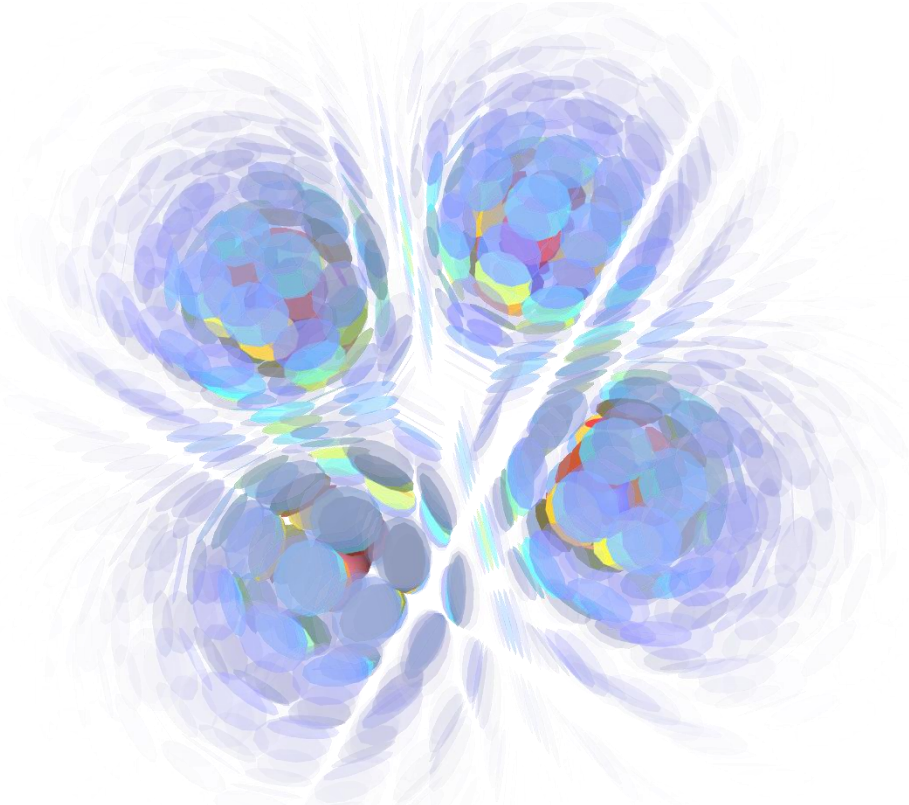
Campo vectorial: +Q [-1.5 0 0], +Q [1.5 0 0], +Q [0 0 -1.5], +Q [0 0 1.5]



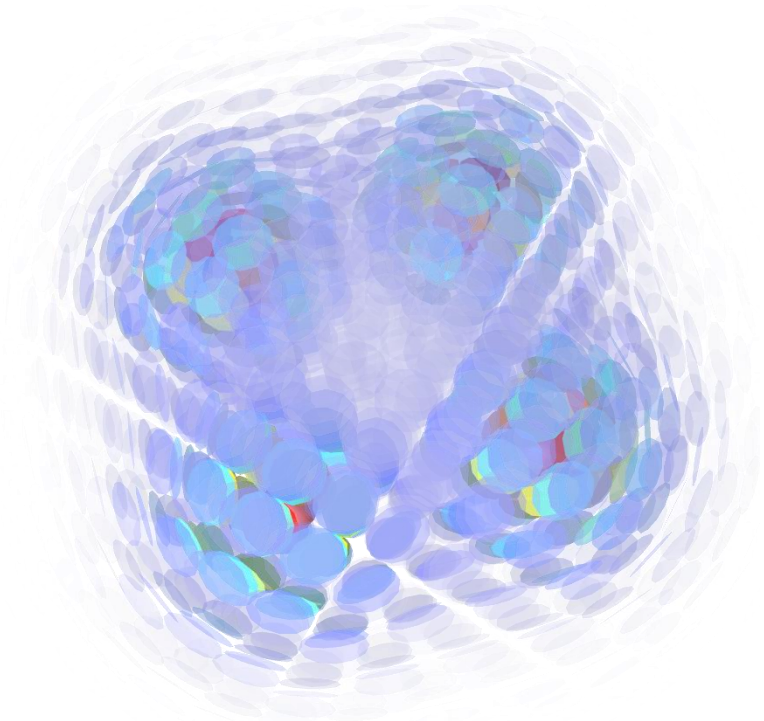
Campo vectorial: -Q [-1.5 0 0], -Q [1.5 0 0], +Q [0 0 -1.5], +Q [0 0 1.5]



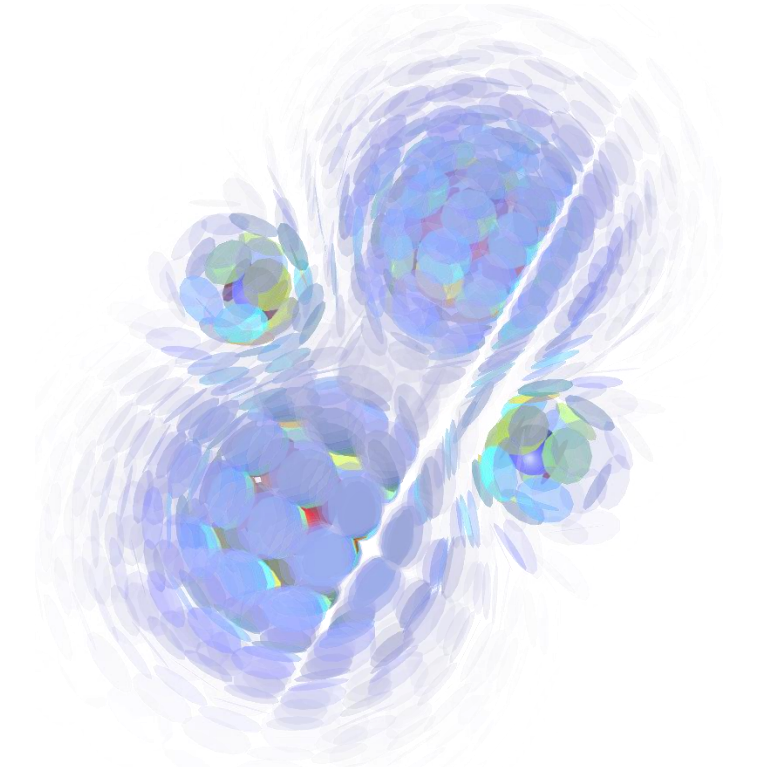
Campo vectorial: -Q [-1.5 0 0], -Q [1.5 0 0], +3Q [0 0 -1.5], +3Q [0 0 1.5]



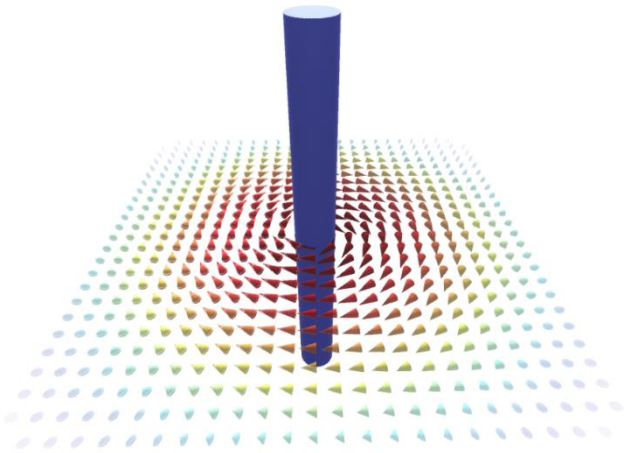
Superficie equipotencial: +Q [-1.5 0 0], +Q [1.5 0 0], -Q [0 0 -1.5], -Q [0 0 1.5]



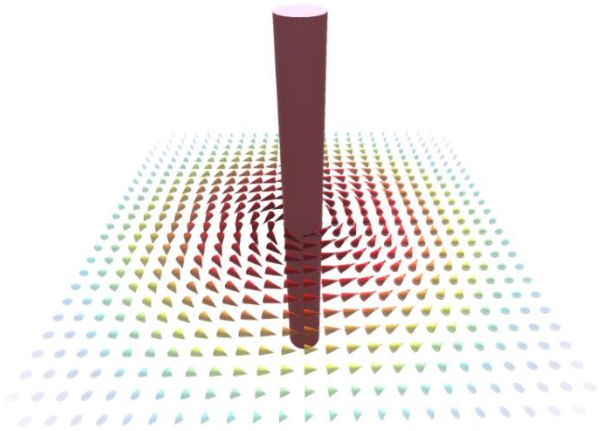
Superficie equipotencial: +Q [-1.5 0 0], +Q [1.5 0 0], +Q [0 0 -1.5], +Q [0 0 1.5]



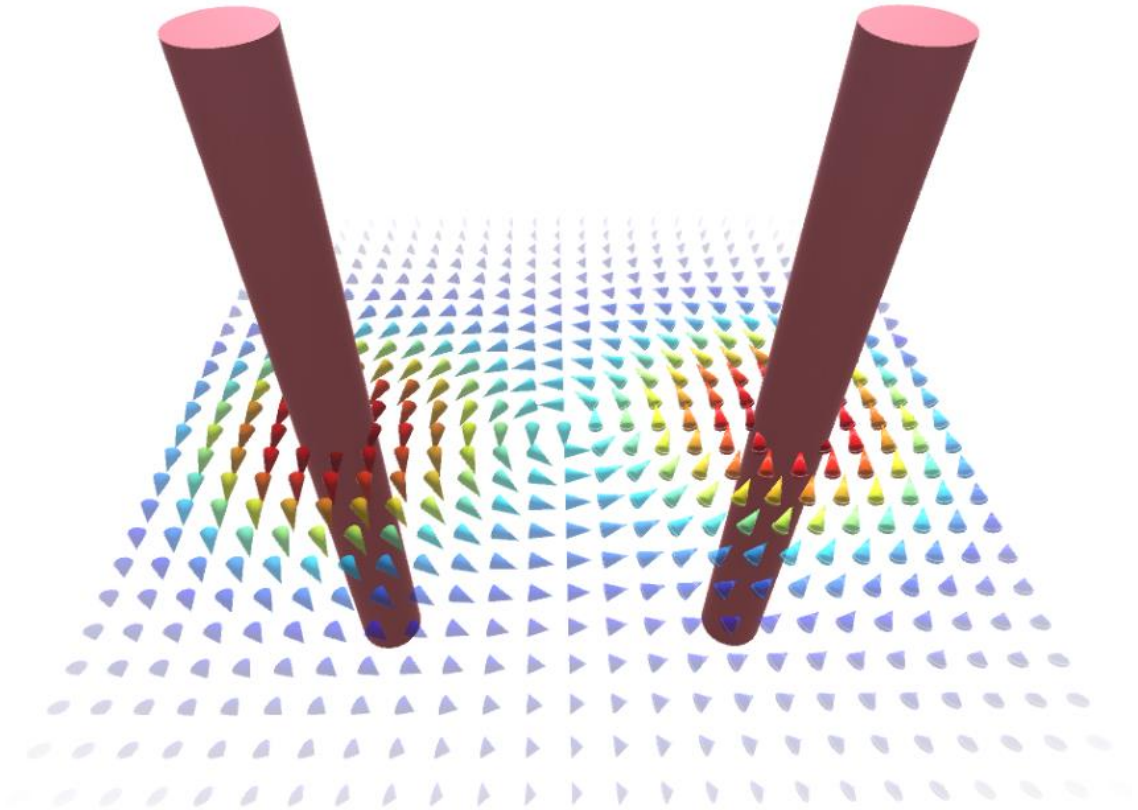
Superficie equipotencial: -Q [-1.5 0 0], -Q [1.5 0 0], +3Q [0 0 -1.5], +3Q [0 0 1.5]



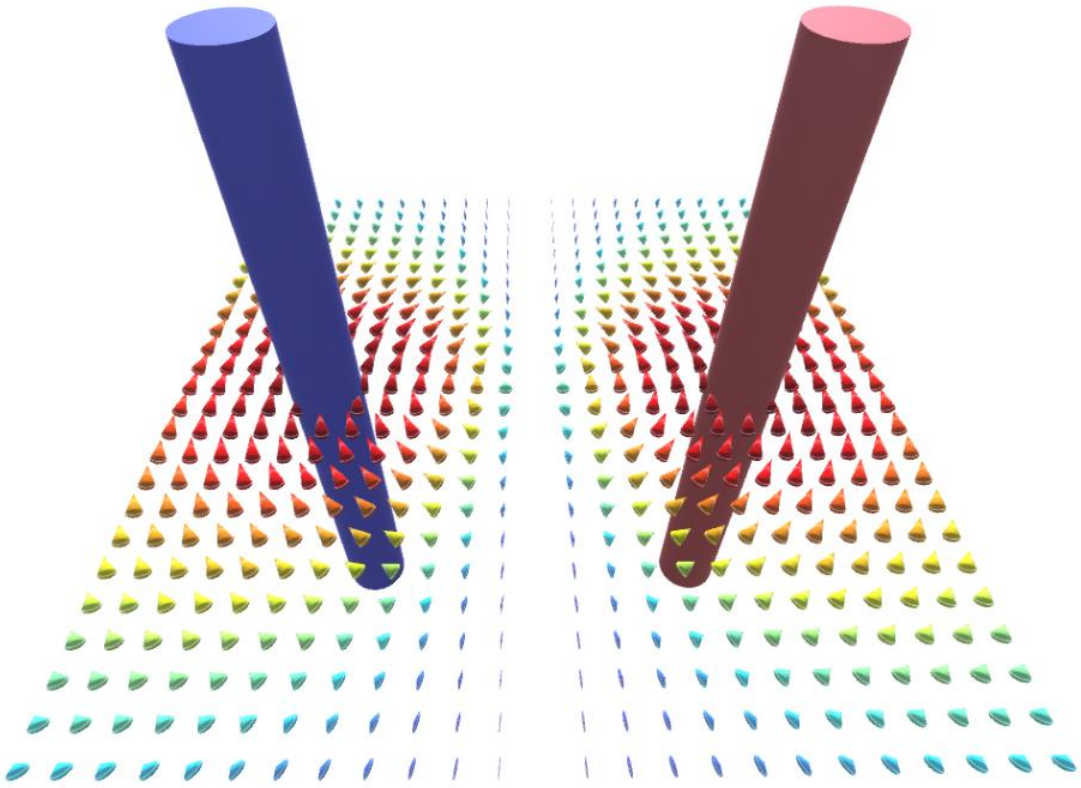
Campo vectorial: Cable con corriente +1A en [0 0 0]



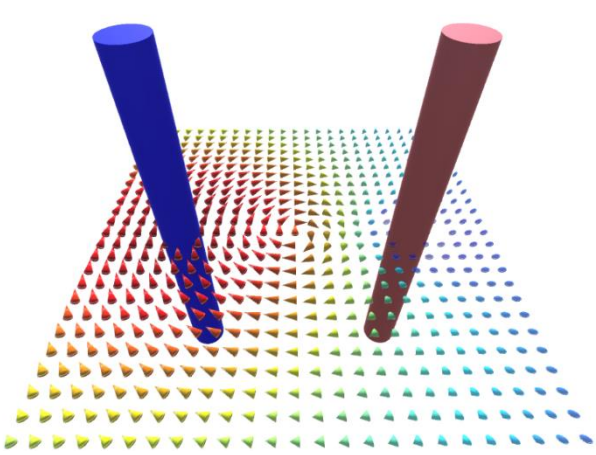
Campo vectorial: Cable con corriente -1A en [0 0 0]



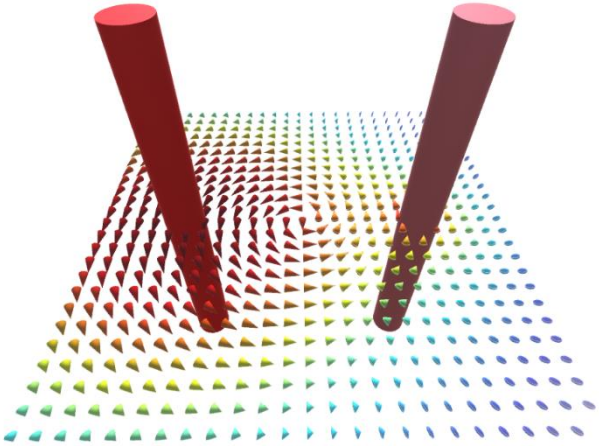
Campo vectorial: Cable con corriente +1A en [-3 0 0], +1A en [3 0 0]



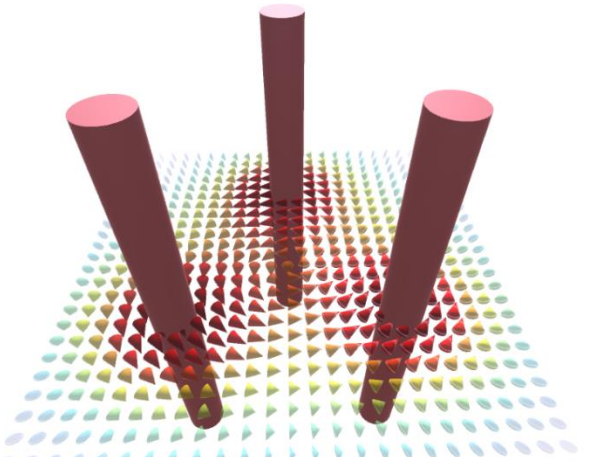
Campo vectorial: Cable con corriente -1A en [-3 0 0], +1A en [3 0 0]



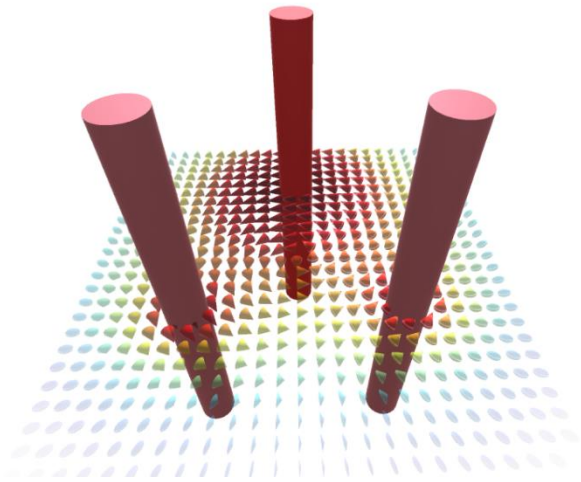
Campo vectorial: Cable con corriente -3A en [-3 0 0], +1A en [3 0 0]



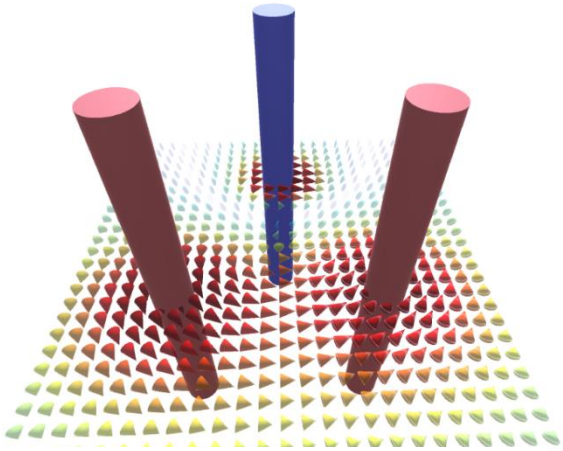
Campo vectorial: Cable con corriente +3A en [-3 0 0], +1A en [3 0 0]



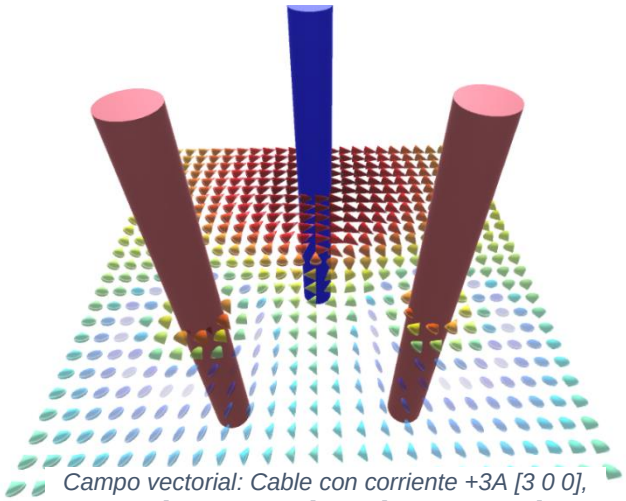
Campo vectorial: Cable con corriente +1A [3 0 0], +1A [-1.5 0 2.5981], +1A [-1.5 0 -2.5981]



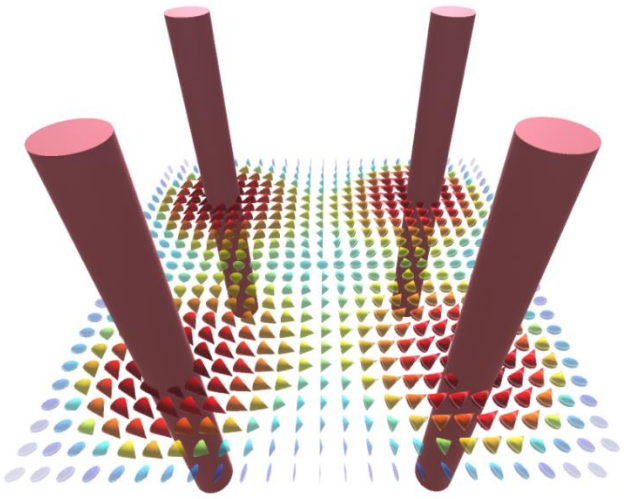
Campo vectorial: Cable con corriente +3A [3 0 0], +1A [-1.5 0 2.5981], +1A [-1.5 0 -2.5981]



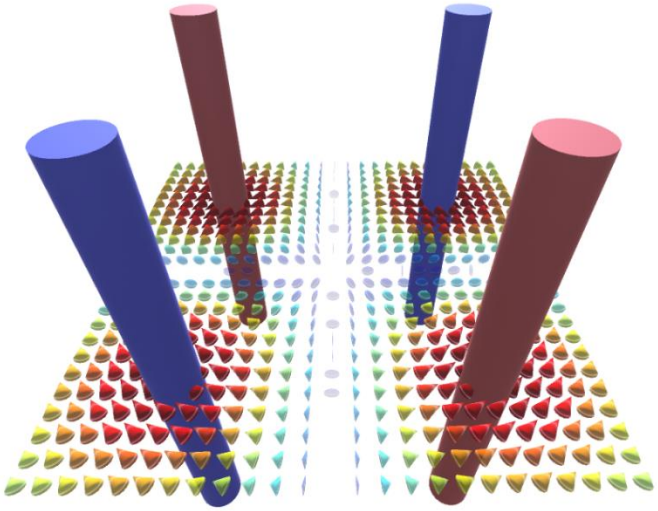
Campo vectorial: Cable con corriente -1A [3 0 0], +1A [-1.5 0 2.5981], +1A [-1.5 0 -2.5981]



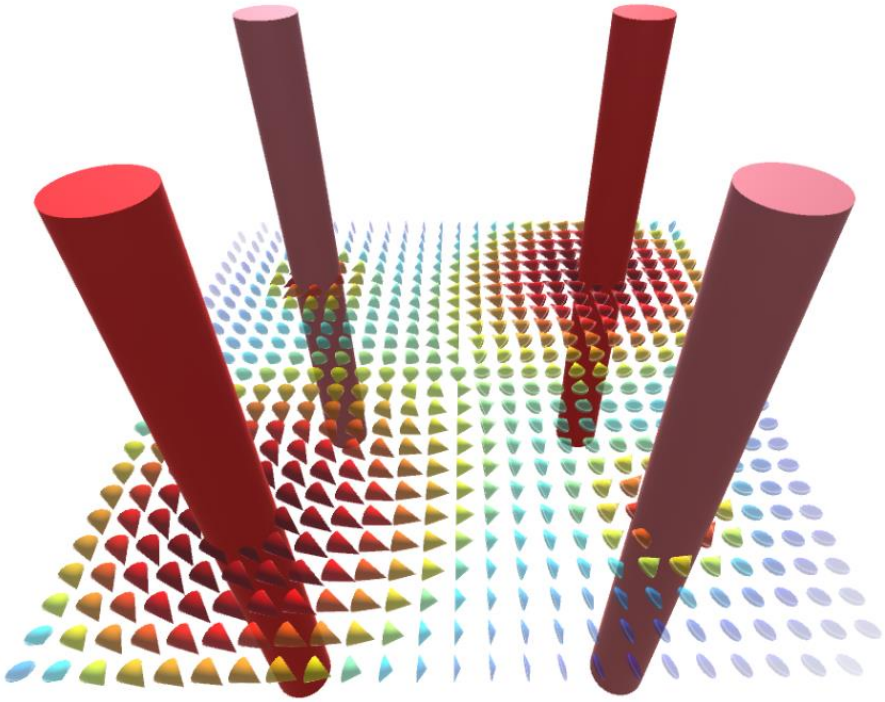
Campo vectorial: Cable con corriente +3A [3 0 0], +1A [-1.5 0 2.5981], +1A [-1.5 0 -2.5981]



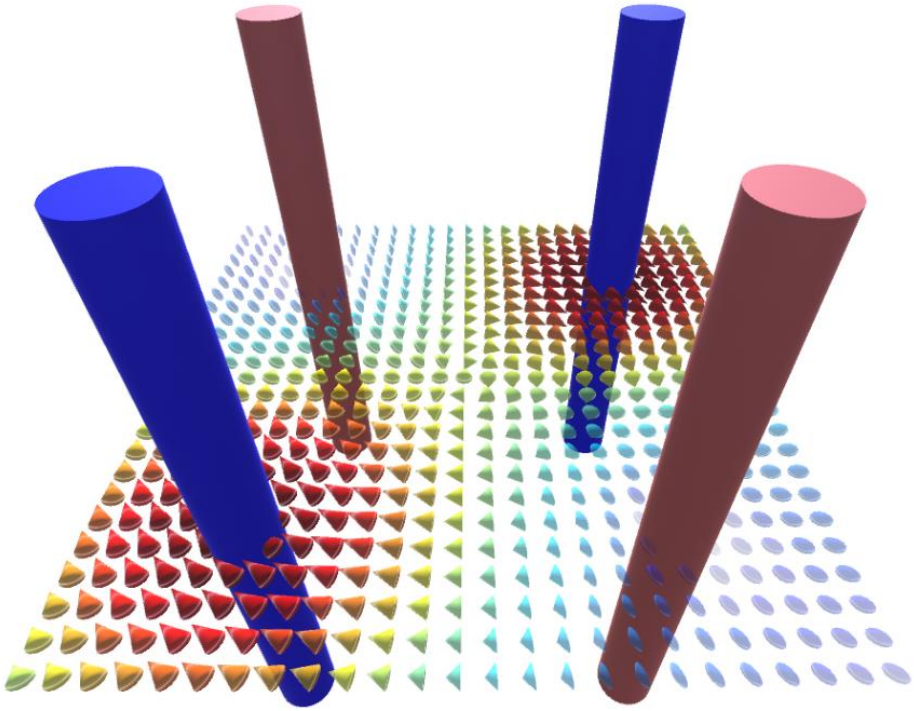
Campo vectorial: Cable con corriente +1A [-1.5 0 -1.5], +1A [-1.5 0 1.5], +1A [1.5 0 1.5], +1A [1.5 0 -1.5]



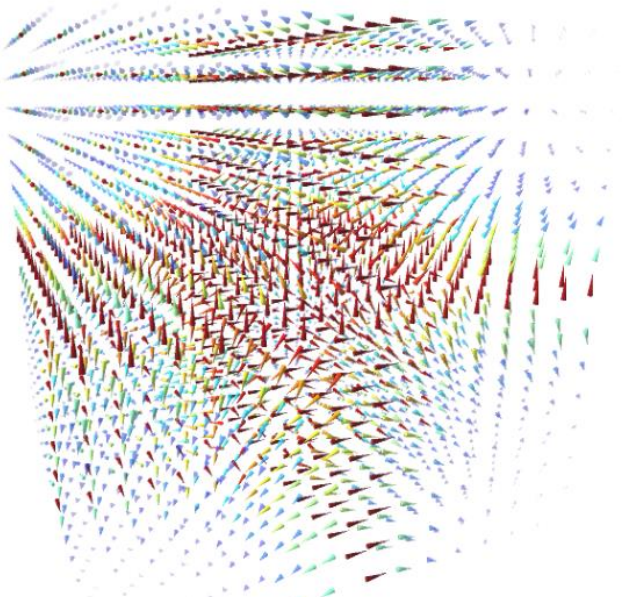
Campo vectorial: Cable con corriente -1A [-1.5 0 -1.5], +1A [-1.5 0 1.5], -1A [1.5 0 1.5], +1A [1.5 0 -1.5]



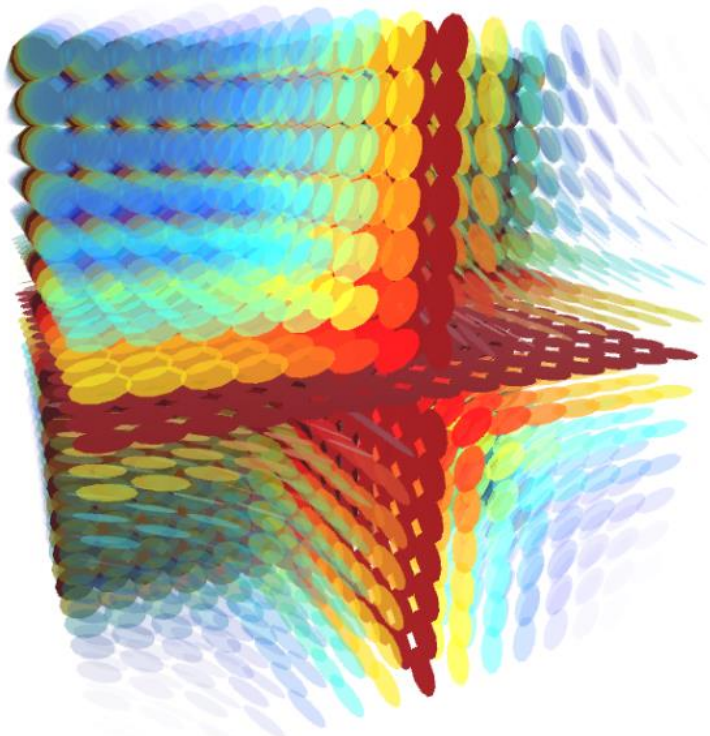
Campo vectorial: Cable con corriente +3A [-1.5 0 -1.5], +1A [-1.5 0 1.5], +3A [1.5 0 1.5], +1A [1.5 0 -1.5]



Campo vectorial: Cable con corriente -3A [-1.5 0 -1.5], +1A [-1.5 0 1.5], -3A [1.5 0 1.5], +1A [1.5 0 -1.5]



Campo vectorial: función vectorial F(x,y,z) = [1/x 1/y 1/z]



Superficie equipotencial: función vectorial F(x,y,z) = [1/x 1/y 1/z]

CAPITULO V

CONCLUSIONES Y RECOMENDACIONES:

CONCLUSIONES:

1. Se logró crear una herramienta que permite generar gráficas de elementos vectoriales más completa, eficiente, compatible con diferentes plataformas y personalizable que las herramientas existentes.

2. Se logró crear una herramienta que permite graficar ejes cartesianos que son personalizables en términos de color, forma, visualización, largo de ejes, disposición de ejes, distancia entre subdivisiones, visibilidad de grilla, y etiquetas.

3. Se logró crear una herramienta que permite graficar vectores que se pueden apuntar en coordenadas cartesianas, cilíndricas y esféricas y son totalmente personalizables en términos de color, modelo, norma de renderizado y modelo cuando la norma vale cero.

4. Se logró crear una herramienta que permite graficar operaciones entre vectores como lo son suma, resta y producto cruz. El vector resultante de cualquier operación es dinámico y se actualiza automáticamente en función de los vectores que lo generan; además hereda todos los métodos de un vector sencillo, por lo cual es igualmente personalizable en términos de color, modelo, norma de renderizado y modelo cuando la norma vale cero.

5. Se logró crear una herramienta que permite graficar el ángulo entre dos vectores. El ángulo graficado es dinámico, personalizable, y se actualiza automáticamente en función de los vectores que lo generan.

6. Se logró crear una herramienta que permite graficar el paralelogramo que se genera entre dos vectores. El paralelogramo graficado es dinámico, personalizable y se actualiza automáticamente en función de los vectores que lo generan.

7. Se logró crear una herramienta que permite graficar un plano referente a un vector. El plano graficado puede ser paralelo o perpendicular al vector, es dinámico, personalizable y se actualiza automáticamente en función de los vectores que lo generan.

8. Se logró crear una herramienta que permite graficar campos vectoriales de una manera muy dinámica, sencilla y personalizable en términos de colormap, modelo de vector, norma de renderizado, modelo cuando la norma vale cero, límites y dimensiones; así como también generar objetos generadores de campo personalizables, con su propia ecuación de campo y modelo.

RECOMENDACIONES:

- Se recomienda para desarrollos posteriores optimizar el desarrollo de gráficas de campos vectoriales cambiantes en el tiempo, así como también mejorar la forma de graficar superficies equipotenciales.

- Se recomienda generar una clase para graficar ejes cilíndricos y esféricos.

- Se recomienda desarrollar una aplicación que haga uso de la tecnología más moderna en realidad aumentada para expresar mejor los conceptos de la matemática vectorial.