CSharpCalc v1.0

Calculation, Simulation, Visualization, Prototyping

3D Graphics in CSharpCalc

An introduction to the CSC3D rendering engine

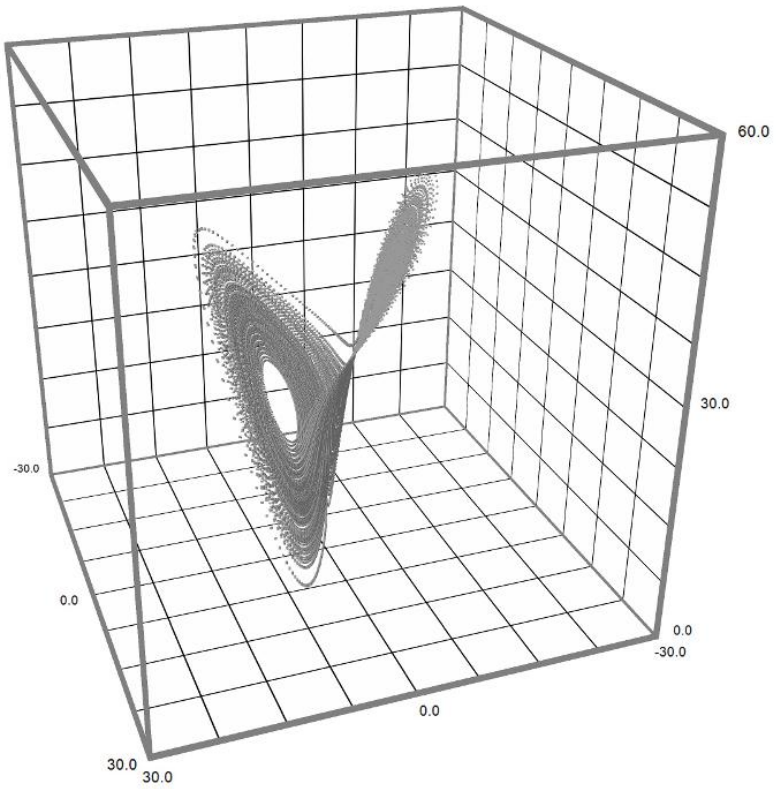


Table of Contents

Terms of Use

Preamble

This document covers selected topics concerning the use of the CSharpCalc software. It is in no way complete and must not be understood as an exhaustive documentation of CSharpCalc. This manual is part of the CSharpCalc software and must not be distributed as a separate document. The script code presented in this manual is provided under the regulations stated in the CSharpCalc license agreement.

Permission to use

You may download and use this document as part of the CSharpCalc software free of charge but without any warranty or liability. If you use this document you accept full responsibility and liability for all consequences of malfunction or faulty results caused by using any information contained in this documentation including, but not limited to, the sample scripts.

Disclaimer of warranty

There is no warranty for the information contained in this manual to the extent permitted by applicable law. This document is provided 'as is'. This includes, but is not limited to, the fitness for any particular purpose. The entire risk of any consequences of errors or ambiguities in this document is with you. Should parts of the information contained in this manual prove false, misleading, inaccurate or incomplete you assume the full cost of all related consequences.

Limitation of Liability

In no event, unless required by applicable law, shall the author of this manual be liable for any damage arising from using information contained in this document, not even if the author has knowledge of the possibility of such damage. Possible damage includes, but is not limited to, loss of data, wrong or inaccurately rendered results or the failure of CSharpCalc (the software covered by this manual) to operate with any other programs.

Distribution of this document

This document is bundled with the CSharpCalc software release it pertains to. Consequently, it is neither intended nor permitted to be distributed as a separate document. If you wish to distribute CSharpCalc, you may use the link:

http://www.csharpcalc.org/download.php

for offering downloads of the CSharpCalc package, including this manual, on your web site.

Governing law

This license agreement shall be governed by the laws of the Republic of Austria. Venue of jurisdiction is Graz, Austria.

Severability clause

In the case that any part of this license agreement is found to be invalid, the validity and legality of all remaining provisions stated in this license agreement shall not be affected or impaired thereby.

Introduction

This document is an introductory text on how to visualize simulation results and scientific data using the CSharpCalc 3D extension. The 3D extension is based on the Windows presentation foundation as included with .net. Being specialized on data visualization, the CSharpCalc 3D extension offers a variety of features, but is also limited in functionality.

Features:

- Construction and rendering of custom 3D objects
- Construction of 3D objects from raw triangle data for importing 3D geometries from external modeling software.
- Texture mapping and shading of surfaces.
- Support of procedural textures.
- Procedural object generation directly from CSharpCalc data structures such as CSCRealVectorFunction.
- Grouping of objects.

Limitations

- No support for advanced 3D geometry construction
- No hierarchic structuring of objects
- No advanced rendering such as ray tracing
- No interaction with 3D geometries - visualization only.

Within this text it is assumed that the reader is familiar with CSharpCalc and the CSharpCalc programming manual. Also detailed knowledge of drawing 2D graphics with CSharpCalc is required as the 3D extension builds essentially on the concepts used for 2D graphics. In addition to knowledge of CSharpCalc, the reader will require knowledge of 3D graphics programming principles. In particular, scene graphs, triangle meshes, texture mapping, geometric transformations, surface materials and illumination are used throughout the text without further explanation. In order to acquire this knowledge the interested reader is referred to freely available computer graphics primers with emphasis on programming, not theory. Detailed knowledge of the rendering pipeline or complex geometry, such as inverse kinematics, is not required for using the CSharpCalc 3D engine.

The CSharpCalc 3D extension consists of only six classes: CSC3DScene, CSC3DObject, CSC3DObjectGroup, CSC3DFactory, CSC3DTexture and CSC3DTextureStyle.

The class CSC3DScene represents the scene graph. It is a static singleton class. During the scene construction process 3D objects, represented by instances of CSC3DObject and

CSC3DObjectGroup, are added to the scene. Once the scene construction is complete, calling CSC3DScene::Render will render the view. In addition to scene graph management, CSC3DScene also encapsulates the functionality for defining the camera attitude as well as for specifying the scene illumination with one directional light and one ambient light.

The class CSC3DFactory provides an ample collection of methods for creating 3D objects ranging from primitives such as cubes and spheres to complex data representations such as space curves and height fields.

Instances of the class CSC3DObject represent triangle mesh objects. Texture coordinates may or may not be available, depending on the chosen object construction method. CSC3DObjects can be moved, rotated and scaled using appropriate methods. Moreover, CSC3DObjects can be grouped by adding them to an instance of CSC3DObjectGroup.

The class CSC3DTexture represents a texture to be used for texture mapping. Textures can be loaded from images located below the Data Directory or they can be created procedurally from within scripts. When rendered procedurally, the class CSC3DTextureStyle defines a collection of parameters such as foreground and background color or a line width.

Due to the large number of functions provided by the CSharpCalc 3D engine, this introduction can only cover a subset of topics related to 3D rendering and data visualization in CSharpCalc. The interested reader is once more referred to the Browser for a full class documentation. All CSC3D classes have been added to the class documentation list which, as in the previous versions, is available from within CSharpCalc.

Basic setup

Rendering 3D scenes in CSharpCalc follows the same line as rendering 2D graphics. The following code shows a simple code framework.

```
// 2D surface initialization. This code is present in any graphical output.
// -------------------------------------------------
CSCDisplay.SetSurfaceSize(1000, 800);
CSCDisplay.SetPhysicalSize(-1, -1, 1, 1);
CSCDisplay.Clear(Color.Black);

// 3D scene setup. This code positions the camera and sets up lights.
// -------------------------------------------------
// Place the camera at position (5.0, 5.0, 5.0) in global coordinates
CSC3DScene.SetCameraPosition(5.0, 5.0, 5.0);

// Let the camera point towards the origin of the global coordinate system
CSC3DScene.SetCameraLookAt(0.0, 0.0, 0.0);

// Enable directional lighting
CSC3DScene.SetDirectionalLightColor(255, 255, 255);
CSC3DScene.SetLightDirection(-0.5, -0.5, -1.5);
```

```
// Disable ambient lighting
CSC3DScene.SetAmbientLightColor(0, 0, 0);

// 3D scene construction
// ------------------------------------------------
CSC3DScene.BeginConstruction();

// Enter your scene construction code here ..

CSC3DScene.EndConstruction();


// Scene rendering and presentation.
// In this code sequence the CSC3DScene object renders
// the scene in the well known BeginRendering - EndRendering brace.
// Finally the rendering is presented on the surface.
// ------------------------------------------------
CSCDisplay.BeginRendering();
CSC3DScene.Render();
CSCDisplay.EndRendering();
CSCDisplay.Present();
```

The first code block initializes the 2D surface we will render the scene to. There is no difference here to setting up the surface for 2D graphics. In the next block, the 3D scene setup, we define the camera position and look at point. As an alternative to the look at point the class CSC3DScene also provides a method for the specification of the camera direction which is not shown in this example. This block also defines the scene illumination. In CSharpCalc 3D only one ambient light and one directional light exists.

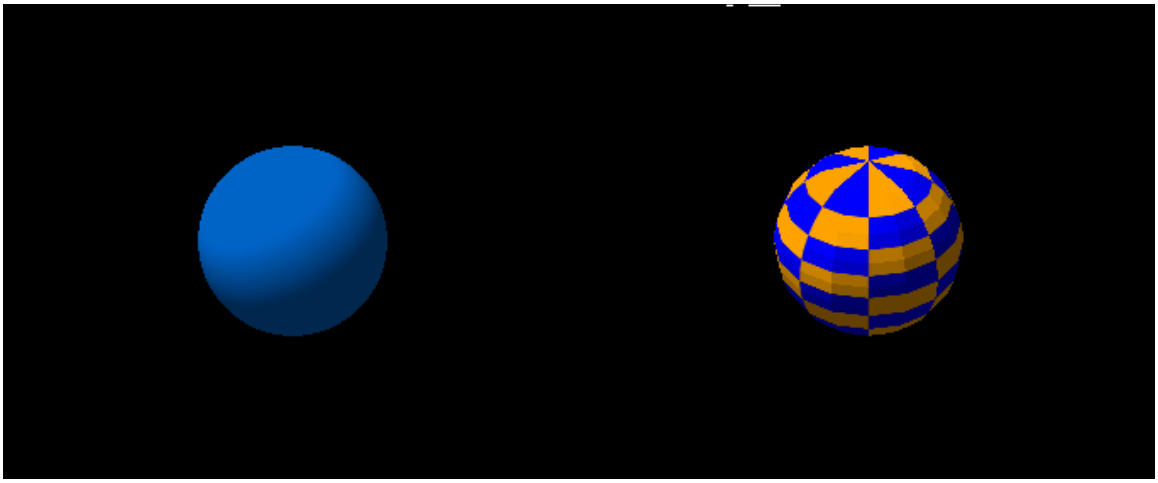
The actual scene construction is braced by two calls, CSC3DScene.BeginConstruction() and CSC3DScene.EndConstruction(). This is very similar to rendering 2D graphics. The above code example does not construct an actual 3D scene, hence the code area for doing that only contains a comment.

The final code block renders the scene and presents it on the surface. As in the case of 2D graphics, rendering takes place between CSCDisplay.BeginRendering() and CSCDisplay.EndRendering(). Calling CSC3DScene.Render() does the actual 3D rendering.

The first scene

The above example showed the framework for rendering 3D scenes with CSharpCalc. Now it is time to actually see a 3D object. As a simple example we replace the comment marking the location for the scene construction code by the following code.

```
// Use the factory to create a sphere...
CSC3DObject SPHERE = CSC3DFactory.CreateSphere(0.5, 64, 64);

// Specify the sphere's ambient surface material ..
SPHERE.SetDiffuse(0, 100, 200);

// .. and add the sphere to the scene
CSC3DScene.Add(SPHERE);
```



A simple sphere and a flat shaded, textured sphere (example below)

This code creates a sphere, defines a diffuse surface material for it and adds the sphere object to the scene.

A flat shaded, textured sphere

In order to replace the single surface color by a texture image we need to redefine the diffuse property of the sphere to be a texture. In order to achieve this, we replace the simple construction code shown in the previous example by the following code:

```
// Use the factory to create a flat shaded, textured sphere...
CSC3DObject SPHERE = CSC3DFactory.CreateSphere(0.5, 16, 16);

// Create a checkerboard texture. First define a style, then create the texture
// using that style.
CSC3DTextureStyle style = new CSC3DTextureStyle(Color.Orange, Color.Blue, 3);
CSC3DTexture TEX = new CSC3DTexture(CSC3DTextureType.Checker,
CSC3DTextureSize.Medium, style);

// Specify the sphere's ambient surface material to be this texture
SPHERE.SetDiffuse(TEX);

// Apply a flat shading transformation ..
SPHERE.Flatten();

// .. and add the sphere to the scene
CSC3DScene.Add(SPHERE);
```

This code is similar to the previous sphere example, but it replaces the simple blue color by a procedurally generated texture image. Texture generation proceeds in two steps. In the first step a style object is created and configured appropriately. Texture styles are represented by instances of CSC3DTextureStyle. The actual texture is created by passing a type, a size and a style to the constructor of the class CSC3DTexture. In order to patch the texture image onto a 3D object's surface it is specified to be this object's diffuse material.

By default, object surfaces in CSC3D are rendered using normal interpolation which gives these surfaces a smooth appearance. Calling the method CSC3DObject.Flatten() converts the object's surface to a flat surface. This conversion is not reversible!

Geometric object transformations

Instances of CSC3DObject can be moved, rotated and scaled. More complex transformations like elastic deformation are not supported by CSharpCalc 3D, but possible if you write your own code. This example shows how to apply rigid transformations

```
// Use the factory to create a cube of size 1 x 1 x 1
CSC3DObject MASTERCUBE = CSC3DFactory.CreateCube();

// scale the master cube by a factor 0.5, i.e. make it half as big.
MASTERCUBE.Scale(0.5);

// Specify the ambient surface material
MASTERCUBE.SetDiffuse(200, 0, 0);

// Create two clones of the master cube and position them in two different
// points. One cube is also rotated.
CSC3DObject CUBE1 = MASTERCUBE.Clone();
CUBE1.Rotate(0.0, 0.0, 1.0, 45.0);
CUBE1.MoveTo(0.5, -0.5, 0.0);
CSC3DScene.Add(CUBE1);

CSC3DObject CUBE2 = MASTERCUBE.Clone();
CUBE2.MoveTo(-0.5, 0.5, 1.0);
CSC3DScene.Add(CUBE2);
```

This code first generates a master cube to be used as a template for the two cubes which are actually added to the scene. Using the method CSC3DObject.Clone(), two cubes are created and subsequently moved and rotated.

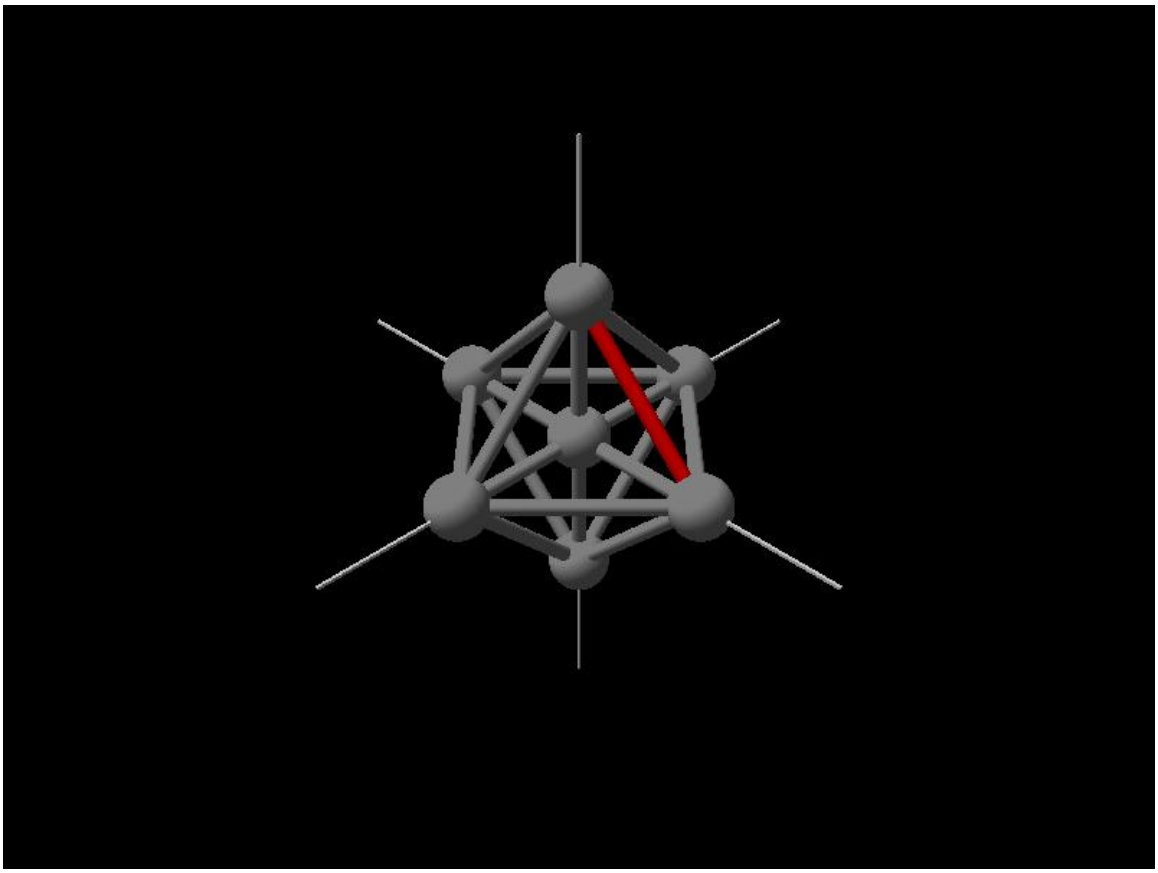
In order to structure scenes CSharpCalc 3D offers the class CSC3DObjectGroup for grouping 3D objects. Group objects can be transformed like single objects, only in this case the transformations are applied to all objects contained in the group in a consistent manner. The following code example illustrates grouping.

```
// Use the factory to create a cube of size 1 x 1 x 1
CSC3DObject MASTERCUBE = CSC3DFactory.CreateCube();
MASTERCUBE.Scale(0.5);

// Create an empty object group
CSC3DObjectGroup CUBEGROUP = new CSC3DObjectGroup();

// specify the ambient surface material
MASTERCUBE.SetDiffuse(200, 0, 0);

// create two clones of the master cube and position them in two points
CSC3DObject CUBE1 = MASTERCUBE.Clone();
CUBE1.Scale(0.5, 0.5, 1.0);
CUBE1.MoveTo(0.5, -0.5, 0.0);

// add the first cube to the group
CUBEGROUP.Add(CUBE1);

CSC3DObject CUBE2 = MASTERCUBE.Clone();
CUBE2.Scale(0.5, 0.5, 1.0);
CUBE2.MoveTo(-0.5, 0.5, 0.0);

// add the second cube to the group
CUBEGROUP.Add(CUBE2);

// now transform the entire group
CUBEGROUP.MoveTo(0.0, 0.0, -1.0);

// finally add the entire group to the scene
CSC3DScene.Add(CUBEGROUP);

// for better orientation also add a colored coordinate cross to the scene
CSC3DObjectGroup COORDS = CSC3DFactory.CreateCoordinateCross(2.0, 0.05, 1.0,
0.0);
CSC3DScene.Add(COORDS);
```

This code creates an instance of CSC3DObjectGroup and adds the two cubes to the group. Instead of adding each cube individually to the scene, only the group containing them is added. Please note that, in addition to transforming the whole group, each object contained can also be transformed individually as well.

A complex 3D scene

Using the knowledge about 3D graphics in CSharpCalc we gained so far, we can now attempt to model and render a more complex scene. As an example we shall construct a molecular structure, also resembling modern architecture. In our simplified view we assume that a molecule is comprised of balls and sticks. Balls will be represented by spheres, while sticks will be represented by cylinders. Both primitives can be generated using the Factory.



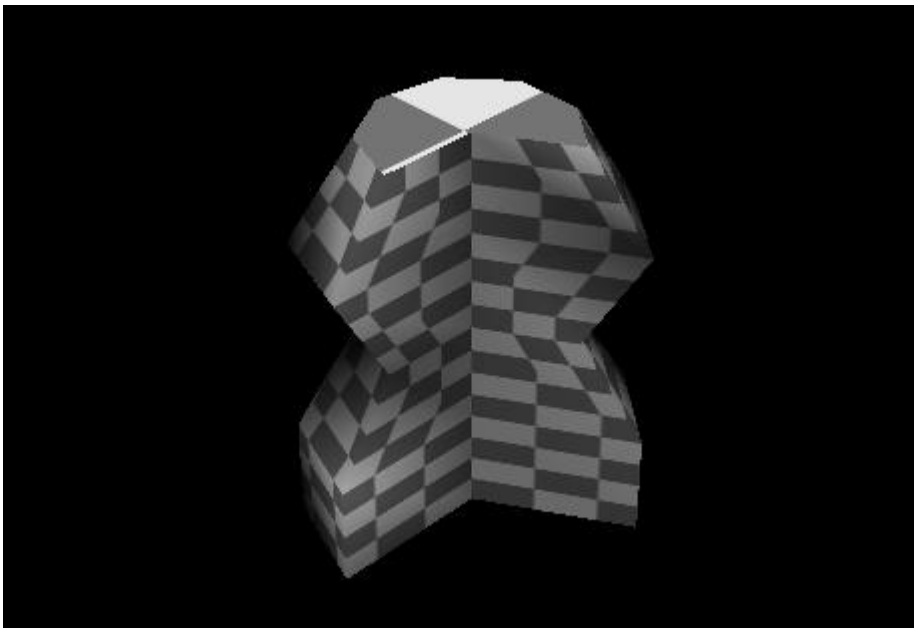
In order to store the entire molecule as one single entity, we use an instance of CSC3DObjectGroup. Once this is done we add clones of our balls and sticks to this group one after the other. The sticks are stretched between atom positions using the method CSC3DFactory.CreateObject(..). This method is useful for creating a clone of an object which is fitted between the two points passed. In order to illustrate the use of the subscript operator of object groups, we finally color one of the sticks red. The entire construction code reads as follows.

```
// Create an object group named ATOMS
CSC3DObjectGroup ATOMS = new CSC3DObjectGroup();

// We shall use this reference for object clones
CSC3DObject OBJ = null;

// Create a sphere for representing the atoms
CSC3DObject SPHERE = CSC3DFactory.CreateSphere(0.2, 32, 32);

// Create a cylinder for representing the connections between atoms.
CSC3DObject CYLINDER = CSC3DFactory.CreateCylinder(0.05, 2, 32);

// Add 7 atoms to the group
OBJ = SPHERE.Clone(); OBJ.MoveTo(0.0, 0.0, 0.0); ATOMS.Add(OBJ);
OBJ = SPHERE.Clone(); OBJ.MoveTo(1.0, 0.0, 0.0); ATOMS.Add(OBJ);
OBJ = SPHERE.Clone(); OBJ.MoveTo(-1.0, 0.0, 0.0); ATOMS.Add(OBJ);
OBJ = SPHERE.Clone(); OBJ.MoveTo(0.0, 1.0, 0.0); ATOMS.Add(OBJ);
OBJ = SPHERE.Clone(); OBJ.MoveTo(0.0, -1.0, 0.0); ATOMS.Add(OBJ);
```

```
OBJ = SPHERE.Clone(); OBJ.MoveTo(0.0, 0.0, 1.0); ATOMS.Add(OBJ);
OBJ = SPHERE.Clone(); OBJ.MoveTo(0.0, 0.0, -1.0); ATOMS.Add(OBJ);

// Create cylinders spanned between the atom positions to connect the atoms
// Connections to center
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0);
ATOMS.Add(OBJ);

// Side connections in the xy plane
OBJ = CSC3DFactory.CreateObject(CYLINDER, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 1.0, 0.0, -1.0, 0.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, -1.0, 0.0, 0.0, 0.0, -1.0, 0.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, -1.0, 0.0, 1.0, 0.0, 0.0);
ATOMS.Add(OBJ);

// Side connections to upper atom
OBJ = CSC3DFactory.CreateObject(CYLINDER, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, -1.0, 0.0, 0.0, 0.0, 0.0, 1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, -1.0, 0.0, 0.0, 0.0, 1.0);
ATOMS.Add(OBJ);

// Side connection to lower atom
OBJ = CSC3DFactory.CreateObject(CYLINDER, 1.0, 0.0, 0.0, 0.0, 0.0, -1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, -1.0, 0.0, 0.0, 0.0, 0.0, -1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, 1.0, 0.0, 0.0, 0.0, -1.0);
ATOMS.Add(OBJ);
OBJ = CSC3DFactory.CreateObject(CYLINDER, 0.0, -1.0, 0.0, 0.0, 0.0, -1.0);
ATOMS.Add(OBJ);

// Assign a different material to one of the objects
ATOMS[19].SetDiffuse(200, 0, 0);
CSC3DScene.Add(ATOMS);

// For better orientation also add a coordinate cross to the scene
CSC3DObjectGroup COORDS = CSC3DFactory.CreateCoordinateCross(2.0, 0.02);

CSC3DScene.Add(COORDS);
```

Paths and extrusion objects

So far we only used spheres and cylinders for the construction of 3D scenes. However, CSharpCalc is capable of constructing more complex objects. One important class of complex objects are so-called extrusion objects. An extrusion object is created by raising a path, for example an arc, which is confined to the XY plane into the Z direction.



An extrusion object constructed by elevating a path in the XY-plane. Each ring of the extrusion has been given a different radius.

In order to construct this object in CSharpCalc we need a 2D path to elevate. The class CSC3DFactory offers a number of methods to create such path objects. Path objects are represented by the class CSC3DObject but they contain no triangle definition, only a sequence of points. A path is called closed if the last point in the sequence has the same position as the first point. In order to construct a closed elevation object, one needs to elevate a closed path. However, being closed is not a necessary condition for a path to be useful for elevation.

Once a path object is available it can be used to construct an elevation object by passing it to the appropriate method of CSC3DFactory. In order to modulate the elevation a sequence of radii can be specified along with the path object. The resulting elevation object will have as many rings as there are elements in this radius array and each of these rings will have the radius specified by the corresponding array element.

In order to add caps to the elevation surface one needs to generate an area object from the path defining the object's outline. If the path is convex or star-shaped the class CSC3DFactory has methods for generating such areas. If not you need to construct the caps yourself.

Once all ingredients for the construction are available they are passed to the CSC3DFactory which in turn returns the final elevation object. The subsequent code illustrates the process, which is actually much simpler than its description.

```
// Create a closed circle segment to be used as extrusion object.
CSC3DObject BaseOBJ = CSC3DFactory.CreateCircleSegment(1.0, 90.0, 330.0, 6,
0.0, 0.0, 0.0);

// Convert the circle segment to a convex area object to be used as caps.
CSC3DObject CapOBJ = CSC3DFactory.CreateConvexArea(BaseOBJ);

// Create a sequence of radii for modulating the extruded object's rings
double[] R = new double[] { 0.8, 0.8, 0.5, 0.8, 0.5 };

// Using the above information, create the extruded object and move it.
CSC3DObjectGroup G = CSC3DFactory.CreateExtrudedObject(BaseOBJ, CapOBJ, 2.0,
R);
G.MoveRel(0.0, 0.0, -1.0);

// Apply a procedural texture to the three object parts.
CSC3DTextureStyle style = new CSC3DTextureStyle(Color.Gray, Color.White, 4);
G[0].SetDiffuse(new CSC3DTexture(CSC3DTextureType.Checker,
CSC3DTextureSize.Small, style));

// Change the texture style.
style.Level = 1;
G[1].SetDiffuse(new CSC3DTexture(CSC3DTextureType.Checker,
CSC3DTextureSize.Small, style));
G[2].SetDiffuse(new CSC3DTexture(CSC3DTextureType.Checker,
CSC3DTextureSize.Small, style));

// Add the extruded object to the scene.
CSC3DScene.Add(G);
```

The parameters passed to the creation methods are explained in the Browser.

Path objects

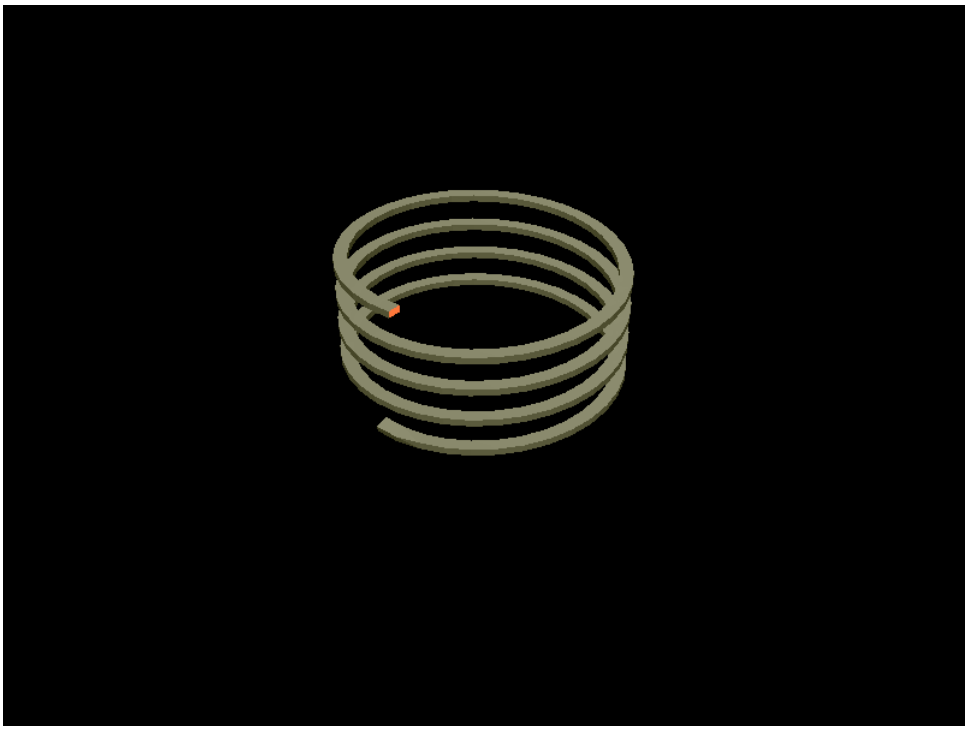
Many applications of scientific visualization require the construction of objects resembling space curves. CSharpCalc offers construction of such objects from instances of CSCRealVectorFunction under the condition that the vectors stored in the vector function have 3 dimensions. In addition to the path the factory also takes a path object, for example a square or a circle, for defining the cross section perpendicular to the space curve. The example below illustrates the construction of a coil shaped space curve with a rectangular cross section.

```
// create a coil-shaped space curve represented as CSCRealVectorFunction
CSCRealVectorFunction coil = Coil.Create();

// Create a rectangular path object defining the cross section
CSC3DObject BaseOBJ = CSC3DFactory.CreateRectangle(0.1, 0.05);

// Create a filled rectangle of the same size to be used for the caps
CSC3DObject CapOBJ = CSC3DFactory.CreatePlane(0.1, 0.05);

// Create the path object representing the coil space curve
CSC3DObjectGroup G = CSC3DFactory.CreatePathObject(BaseOBJ, CapOBJ, coil);
```

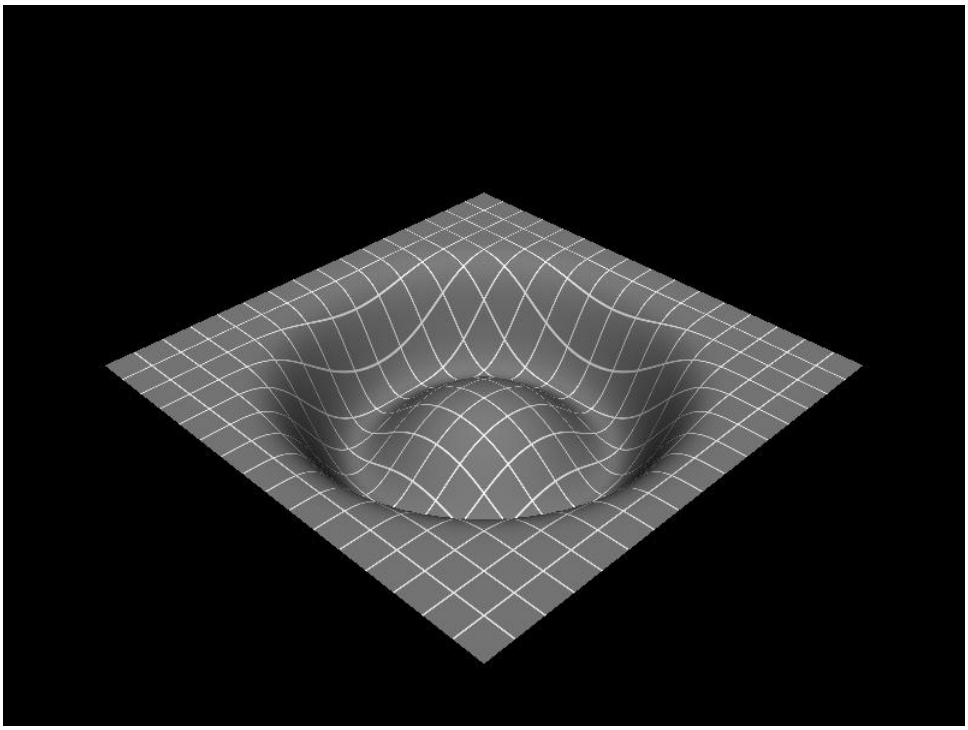
```
// Adapt the coil surface properties. Caps are highlighted in orange color.
G[0].Flatten();
G[0].SetDiffuse(100, 100, 100);
G[0].SetEmissive(50, 50, 20);
G[1].SetEmissive(250, 80, 20);
G[2].SetEmissive(250, 80, 20);

// Finally add the coil object (group) to the scene.
CSC3DScene.Add(G);
```

The coil is procedurally created by the following class:

```
public static class Coil
{
 public static CSCRealVectorFunction Create()
 {
  CSCRealVectorFunction curve = new CSCRealVectorFunction();

  double phi = 10.0;
  double x = 1.0;
  double y = 0.0;
  double z = 0.0;

  double cp = Math.Cos(Math.PI*phi/180.0);
  double sp = Math.Sin(Math.PI*phi/180.0);

  double x0 = x;
  double y0 = y;
  double z0 = z;

  curve.Samples.Add(new CSCRealVector(new double[] {x, y, z}));

  for(int j = 0; j <= 144; j++)
  {
   x = cp*x0 - sp*y0;
   y = sp*x0 + cp*y0;
   z = z0 + 0.25*j*phi/360.0;

   x0 = x;
   y0 = y;

   curve.Samples.Add(new CSCRealVector(new double[] {x, y, z}));
  }
  return curve;
 }
}
```



A coil-shaped path object



A height field resembling a real function of two real variables, i.e. z = f(x, y)

Height fields

Height fields are used for visualizing functions of the form z = f(x, y), but also for geographical information or simply curved shapes. The Factory offers one method to construct a CSC3DObject from a 2D array which is interpreted as height information. The following source code illustrates the construction of a textured height field with a custom texture style.

```
// Create a style for the texture
CSC3DTextureStyle style = new CSC3DTextureStyle(Color.White, Color.Gray, 4);

// Create a height field object. The height field itself is defined in the
classcode below.
CSC3DObject OBJ = CSC3DFactory.CreateHeightField(4.0, 4.0,
HeightField.Create());

// Specify the texture to be a line grid rendered in the above style.
OBJ.SetDiffuse(new CSC3DTexture(CSC3DTextureType.Grid, CSC3DTextureSize.Large,
style));

// Add the height field object to the scene.
CSC3DScene.Add(OBJ);
```

The height field itself is procedurally created by the class HeightField.

```
// This class generates a height field example.
public static class HeightField
{
 public static double[ , ] Create()
 {
  double[ , ] Z = new double[tiles + 1, tiles + 1];
  for(int j = 0; j <= tiles; j++)
  {
   for(int i = 0; i <= tiles; i++)
   {
    double x = 2.0*(-1.0 + 2.0*i/tiles);
    double y = 2.0*(-1.0 + 2.0*j/tiles);
    double r2 = x*x + y*y;
    Z[i, j] = -0.3*Math.Exp(-1.0*r2*r2 + 2.0*r2);
   }
  }
  return Z;
 }
 const int tiles = 64;
}
```

In order to add additional information, for example the phase angle of a complex function or aerial imagery, to the height field, custom textures may be rendered and applied to the height field geometry.

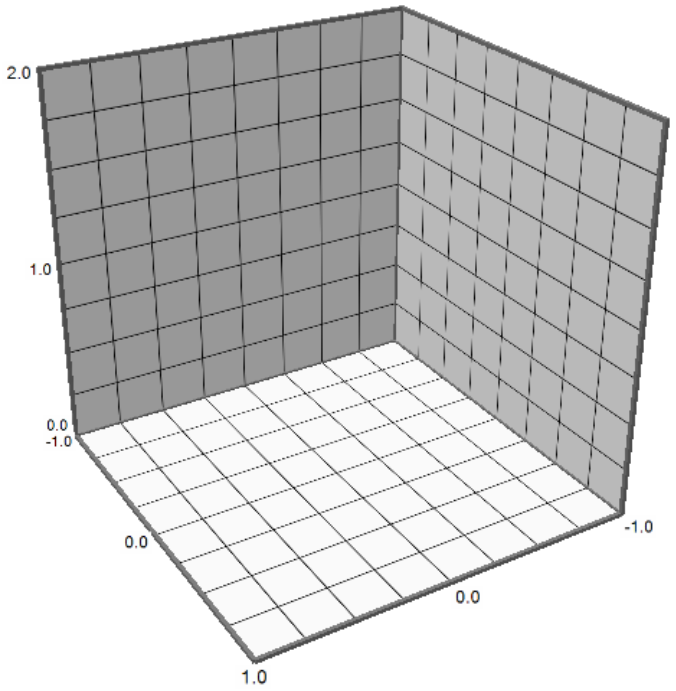
Coordinate Systems in 3D

In order to display 3D data in a numerical context one requires coordinate systems with number labels attached. CSharpCalc offers functions to construct highly customized models of coordinates. The axes and 3D lines are modeled using the available 3D primitives such as cylinders, planes and space curves. Numerical text is added to a model as so-called label texture. A label texture is a special type of procedurally created texture which contains text. The example below explain illustrates their use.

```
// create a style for the textures
CSC3DTextureStyle style = new CSC3DTextureStyle(Color.Black, Color.White, 3, 3,
1.0, 1.0);

// create a grid texture for the back planes
CSC3DTexture gridtex = new CSC3DTexture(CSC3DTextureType.Grid,
CSC3DTextureSize.Large, style);

// also create label textures
CSC3DTexture label00 = new CSC3DTexture("-1.0", CSC3DTextureSize.Medium,
style);
CSC3DTexture label01 = new CSC3DTexture("0.0",  CSC3DTextureSize.Medium,
style);
CSC3DTexture label02 = new CSC3DTexture("1.0",  CSC3DTextureSize.Medium,
style);
CSC3DTexture label03 = new CSC3DTexture("2.0",  CSC3DTextureSize.Medium,
style);

// create the frame box using the factory
CSC3DObjectGroup BOX = CSC3DFactory.CreateOpenFrameBox(0.02, -1.0, -1.0, 0.0,
1.0, 1.0, 2.0,
true);

// Patch the grid textures to the frame box's back planes ..
BOX[0].SetDiffuse(gridtex);
BOX[1].SetDiffuse(gridtex);
BOX[2].SetDiffuse(gridtex);

// .. and add the box
CSC3DScene.Add(BOX);

// Add text labels and position them accordingly.
// Small offsets are added to the positions for improved visual appearance.
CSC3DScene.Add(CSC3DFactory.Create3DLabel(-1.0, 1.12, 0.0, 0.1, label00));
CSC3DScene.Add(CSC3DFactory.Create3DLabel( 0.0, 1.12, 0.0, 0.1, label01));
CSC3DScene.Add(CSC3DFactory.Create3DLabel( 1.05, 1.08, 0.0, 0.1, label02));

CSC3DScene.Add(CSC3DFactory.Create3DLabel(1.1, -1.0, 0.0, 0.1, label00));
CSC3DScene.Add(CSC3DFactory.Create3DLabel(1.1,  0.0, 0.0, 0.1, label01));
```

```
CSC3DScene.Add(CSC3DFactory.Create3DLabel(1.1, -1.0, 0.1, 0.1, label01));
CSC3DScene.Add(CSC3DFactory.Create3DLabel(1.1, -1.0, 1.0, 0.1, label02));
CSC3DScene.Add(CSC3DFactory.Create3DLabel(1.1, -1.0, 2.0, 0.1, label03));
```

The first code lines define a texture style and a grid texture. Subsequently, four label textures are created for displaying the labels "-1.0", "0.0", "1.0" and "2.0". The open frame box is used as basis for the coordinate system. It represents the first octant of a Cartesian 3D coordinate frame. The back planes of the open frame box are textured with a grid pattern. Alternatively one may also use a coordinate tripod, a coordinate cross or a custom model representing the required coordinate frame.



A fully customized 3D coordinate frames with axes, grid and lettering at user-defined positions

The final block generates 3D versions of the labels using the CSC3DFactory.Create3DLabel(..) method. This method automatically rotates the plane with the label texture towards the camera direction (not position!). Hence, it is important that the camera attitude is specified prior to scene construction.

Summary and further reading

This text gave a brief introduction to the CSharpCalc 3D engine. Due to the complexity of the topic a complete presentation of all features of CSharpCalc is neither possible nor desirable, since such a presentation would be confusing and unreadable. For further information on 3D rendering functionality offered by CSharpCalc the esteemed reader is once more directed to the Browser. All classes pertaining to the CSharpCalc 3D extension are named using CSC3D as name prefix.

In addition to the Browser, a repository with example scripts for many applications is currently under construction on csharpcalc.org. Check the web site for further developments.