User manual - CellEngine 0.1

How to compile the library

Linux

Go to “CellEngine” folder, and execute make. The makefile will build the library called “libcellengine.a” into the “build” folder. The compilation is made with `-O3` optimization level by default.

How to compile a program with CellEngine

gcc/g++

Only you need:

- Add `-ICellEngine/include` to the include path.
- Add `-LCellEngine/build` to the lib path
- Add the next directive to link to the library `-lcellengine`

A little example:

```
gcc -o out main.c -ICellEngine/include -LCellEngine/build -lcellengine
```

Programming paradigm

CellEngine follows several rules of programming along its code:

- All fields of the structures are only read. If you want to write some, you must use the proper set function instead.
- All public structures, functions and constants begin with the prefix `ce` following by the name.
- All private global variables, functions and constants start with the prefix `_ce` following the name.
- All types, variables, and functions names begin with lowercase letters and separate the different words contained with uppercase letters. Example: `ceWordWord`
- All constants are totally written in uppercase letters separated by low bars if it is necessary. Example: `CE_WORD_WORD`

Code example

The next code is the minimal example. With these few lines you can have an entire bacteria colony growing.

```
#include "CellEngine.h"
int main(int argc, char** argv)
{
    ceInit(4, 1, 40.0f, 20.0f, 10.0f);
    ceSpace* space = ceCreateSpace();
```

```
    ceCreateBody(space, 20.0f, ceGetVector(0.0f, 0.0f), 0.0f);
    while(1)
    {
        ceBody* body = space->firstBody;
        while(body != NULL)
        {
            ceGrowBody(body, 0.1f);
            if(body->length >= 40.0f)
                ceDivideBody(body, 0.02f, 0.45f);
            body = body->next;
        }
        ceStep(space);
    }
    return 0;
}
```

Let's see how it works.

First, the line `#include "CellEngine.h"` includes all resources from CellEngine, you don't have to import anything more.

The line `ceInit(4, 1, 40.0f, 20.0f, 10.0f);` initializes CellEngine. Let's see each of its parameters.

- The first integer parameter `4` indicates the combined rings that will be used to expand the colony outward. A little value of this parameter will cause physics disconcordance with the reality and high value will need more physics iterations per frame (the next parameter). In the example, we put a 4 because is a medium-high quality parameter (in terms with parallelism with reality) and is not necessary many physics iterations per frame. You can see more about this in the explication of Shove by Rings Algorithm.
- The second parameter is the number of physics iterations per combined ring. This number is proportional to the homogeneous overlap that the colony will have. If you increase the value the overlap will decrease, but the computation will increase too considerable. In the example, `1` means that about 4 physics iterations will be done to the entire colony (number of combined rings multiply by physics iterations per combined rings, you can see more about this in the explication of Shove by Rings Algorithm).
- The third and fourth parameter is the statistical maximum and minimum length that the bacteria will be. It is used for improving the performance of the engine. If any of bacterial exceeds slightly the value put here, nothing happens (in terms of quality).
- The final parameter is the width of all bacteria. In this version of CellEngine all bacteria have the same width, for two reasons: in part for improving the

performance and the memory consumed and because in the reality, all bacteria of the same type, practically have the same width.

The next line: `ceSpace* space = ceCreateSpace();` is simple, we create a new space for contains all bodies and save the returned pointer manage it.

The line `ceCreateBody(space, 20.0f, ceGetVector(0.0f, 0.0f), 0.0f);` creates a body associated to the previous space with length 20, in the position 0 in both x and y coordinates and rotated 0 radians respect x axis.

The next code block represents the loop of the simulation, each iteration of this loop can be related to a visual frame of the program.

This code block contains another one:

```
ceBody* body = space->firstBody;
while(body != NULL)
{
    //...
    body = body->next;
}
```

It will iterate the entire colony, from first to last body. Inside of this code block we have the logic of our physics simulation.

In the line `ceGrowBody(body, 0.1f);` we grow the bodies 0.1 more in the axis of its length.

The next line sets a condition. If the length of the body is greater than 40 units of length the body will be divided by the following line.

The line responsible for the division is `ceDivideBody(body, 0.02f, 0.45f);`. The first parameter is the rotation of the new body respect its father. It is used for creating a little chaos into the division. For more precise experiments, its value must be random between a small negative and small positive (for example between -0.02 and 0.02). The second parameter is the proportion of the length that the child will have after the division. The range goes from 0 to 1.

The important final line is `ceStep(space);`. It will call the engine for computing a step of physics to the entire colony. This call will solve the overlap produced by the growing for each of bacterias moving and rotating all of them.

Free resources

In the previous code example we didn't free any resource.

For removing the space from the memory, you can call to `ceDestroySpace` function. This function will destroy any body attached to the space too.

For removing a single body, you can call to `ceCreateBody` function. This call detach the body from the space and remove the occupied memory of this.

The best efficient way to remove a body from a space and put other with the same properties into another space is calling to `ceMoveBodyToSpace` function.

Getting resources to make the conjugation process

For developing the conjugation mechanism, it is necessary get the bodies around the donor body. You can get this information calling to `ceGetNearBodies` function. This function return a list of the bodies around, at a distance. With the distance parameter you can emulate the pilus length of the bacteria. The range of available values for this parameter goes from 0 to bacteria width.

Once you have the list of the bodies, you will want to do something with that bodies. Likely, you will have your own data for manage the state of the cells, and for each cell, you can relate directly to its body. But, when you get the body list, it is possible that you can't relate the body with the cell easily. For solving this issue, the struct `ceBody` have a field `data` (that you can set with `ceSetData` function to keep the programming paradigm). This field is for user purpose. You can use this field as you like. In the conjugation case, you can use the field data to save a pointer to the cell state, and later, access directly if it's necessary.

Interface utils

CellEngine have two efficient functions to get random access to information from the space. Maybe by the user mouse or by pre programmed areas. They are:

- `ceGetBody` to get a body from a position given by a coordinate of the space.
- `ceGetBodies` to get a list of bodies from a rectangle of the space.