2.6

Cellular Automata for Physical Modeling

Tom Forsyth,

Mucky Foot Productions, Ltd.

tomf@muckyfoot.com

Many current game environments are mostly static. The sorts of things that move in games are restricted to either small, discrete objects, such as vehicles and people, or sometimes some larger, mechanical, or prescripted objects. In some cases, the water level in a container can move in scripted ways, but it is only a single horizontal plane that moves up or down, and there is no way for the player to directly interact with it.

In the current state of the art of games, the following effects tend to be either faked or not simulated at all:

- Fire that spreads, ignites flammable objects, and causes damage to them.
- Water that can be held in containers, flow through pipes, be pumped around realistically, walked through, weigh objects down, overflow containers, or spread over floors and down slopes.
- Oil that combines the fluid properties of water with the burning properties of flammable materials, such as wood.
- Explosions that have realistic damage radii, doing more damage indoors than outdoors, and traveling around corners in realistic ways.
- Heat that causes air to rise, causes convection currents, can be pumped around by ventilation fans, and possibly even carry scents and smells.
- Smoke and dust that spread with air currents, are generated by fires or smoke grenades, obscure vision, and choke people.
- Walls and environments that can be damaged, destroyed, set on fire, moved, or give limited protection from explosions and attacks.

Some of these features have appeared in games, but usually in heavily scripted and constrained ways; frequently they play little part in the actual gameplay and look artificial—which, of course, is exactly what they are. Using cellular automata (CA) to simulate these ideas can lead to far more dynamic and realistic behavior, and allow

new types of gameplay and new tactics within games. At the very least, they allow more realism, better graphical rendering, and therefore increase player immersion.

CA Basics

Cellular automata (CA), and their close relatives, finite element analysis (FEA) and computational fluid dynamics (CFD) [CFD], are already used in plenty of applications for modeling air and water flow, heat distribution, building stresses and strains, and many other aspects of the real world. However, the main emphasis of the academic and commercial modelers is on accuracy. As game programmers, our only real concern is whether something looks good enough and runs quickly enough; and in almost every case, the simulation can be enormously simplified while still looking perfectly correct to most people.

The basics of a CA are simple. The world is divided into a grid of fixed-size cells. Each cell has various numbers associated with it to represent its state. Usual values held in cells are the air pressure, temperature, amount of water, which direction the water or air is flowing in, and so on.

Each game turn, every cell is processed, and it compares itself with its neighboring cells. Differences between them result in changes to the state of the cell and/or its neighbors according to various laws. In this gem, these laws will be based very loosely on real physical laws. One of the best-known CA is called "Conway's Game of Life" [Conway]. This is an extremely simple CA. It has a single bit of state—whether the cell is full or not—and some extremely simple rules for changing state according to the state of neighboring cells. Nevertheless, even this simple model can give rise to some extremely complex behavior.

The CAs used in games will have rules based on various physical models to determine the amount of heat, air, water, or smoke that is transferred between neighboring cells. If we run the rules quickly enough on a sufficient number of cells, water will flow downhill and find level ground, gently heated air will form convection currents, and strongly heated air will burn objects and, in turn, be heated by the burning objects.

In a three-dimensional array of cubic cells, there are three possible definitions of 'neighbor' cells:

- The six cells that share a face with the central cell.
- Those 6, plus another 12 that share an edge with the central cell.
- Those 18, plus another 8 that share a corner with the central cell.

Surprisingly, the rules that are used for physics simulations give almost the same results, whichever of the three definitions we use. Of course, the first version is far simpler, and there is only one type of neighbor cell, rather than three. For this reason, it is far easier to only consider as neighbors the six cells that share a face with the central cell.

First, choose the physical size of a CA cell. For human-size games, we decided to use cubes that are half a meter across. Any bigger, and a CA cell of air will not fit inside a narrow passageway. Smaller cubes give higher resolution and allow for smaller pipes, narrower gaps, and so on—but the extra space and processing is expensive. Different scales of games will naturally require a different size of CA cell; however, because most games are set on a human scale, for convenience, this gem will assume a scale of half-meter cube cells in its examples.

Another important consideration in a human-size game is how to model thin walls. Most internal house walls and doors are only a few centimeters thick. They will stop water flow, slow fire down, and stop smoke and air spreading, so they must be modeled in some way. Modeling them conventionally by using many small cells, and marking those occupied by the wall as solid, would require using cells of no more than about 10 cm across, which requires 125 times as many cells—an extremely expensive option.

Two possible solutions present themselves. One method, used by the first of the *X-Com* series of games in their impressive and innovative use of CAs [XCom], is to model the faces between the cells as entities, as well as modeling the cells themselves. So walls, floors, and ceilings always lie between two cells, along cell faces. This works quite well, but it does mean that there are now two distinct classes of objects—things that fill a whole cube (e.g., rock, dirt, furniture, or tall grass) and things that sit between two cells (e.g., walls, floorboards, short grass, or doors). This creates annoying special cases in the code used to model substances and their interactions, and causes code replication between the two types (spaghetti code). However, if this model fits, then it is a viable one, and it is fairly intuitive—the internal representation of objects matches their rendered shape fairly closely.

The other solution is one that retains its generality without resorting to many tiny cells. Rather than aligning cubic cells on a fixed grid, we allow the edges between cells to move about a bit according to the contents. This allows a thin wall to be chopped into half-meter squares, and each square lives in a cell. Because the walls are only a few centimeters thick, we expand the neighboring cells to take up the extra space. Even though the cells are no longer aligned on a grid, the CA code itself does not know or care what shape the objects it represents are. As far as the CA physics are concerned, everything is still half a meter thick. Most of the work in making things look otherwise is in the rendering, rather than in the CA routines. It is the job of the rendering to ensure that water goes all the way to the wall's mesh and not just to the edge of the CA cube, which would leave a large gap. The only things that need to spread adaptively like this are volumetric effects, such as smoke, fire, and water. When drawing a cell with one of these effects, the renderer needs to check each neighboring cell to see if its polygonal shape is smaller than the usual half-meter cube. If it is, it expands the size of the volumetric effect to fill the space.

In this scheme, a one-meter-wide corridor with thin wooden walls is represented by a plane of 'wood' cells, a plane of 'air' cells, and then a plane of 'wood' cells. Since

the centers of the cells are each half a meter away from each other, the total apparent width from wall to wall is still one meter. Of course, the graphical representation of the world still shows that the 'cubes' of wood are not cubes at all, but flat planes a few centimeters thick; and this is the representation that will be used for any collision detection. But the distinction makes very little difference to the things that are modeled with the CA. Because these entities are fairly amorphous, the difference between what is rendered (a one-meter gap) and what is actually being modeled (a half-meter gap) is very hard for the player to see. Again, accuracy is sacrificed for speed wherever the game can get away with it.

The next factor to consider is a gameplay decision—the difference between using passive scenery and active scenery.

Passive Scenery

In this system, as far as the CA is concerned, the scenery is inert—it is not affected by the actions of the CA in any way. This is the simpler of the two representations, but it still allows discrete objects, such as the ubiquitous oil drum and crate, to float away on rivers of water or to explode or burn when heated by fire.

Because the CA only knows about cells, not polygons, the scenery must be converted into a cell representation—usually as a preprocessing step. These cells are simply marked as inert volumes that confine the actions of the CAs. Of course, the scenery is usually a collection of arbitrary polygons and is not aligned to cell boundaries. But the things being modeled with CAs are so amorphous that this difference does not matter in practice. As long as each solid polygonal wall is converted into a continuous wall of CA cells, water will not flow through and break the illusion.

Even in this system, special cases should be made for doors that can be opened and other animate or moving objects. When doors are opened, they should remove (sliding doors) or move (swinging doors) the solid cells that represent them so that water and/or fire can move through them.

Active Scenery

The far more versatile and adventurous option is to have the scenery modeled by the CA as well. This opens up the 'totally destructible world' concept that many designers are looking to as the next big thing in games, though this concept is not truly new in computer games [XCom].

In this system, rather than simply being cells of inert material, scenery is modeled by its actual properties, such as temperature, flammability, and so on. As the cells modeled with CA change their state according to the physical rules of the CA, the graphics engine changes how it renders the associated polygonal objects (e.g., sooty, damaged, etc.)

In the latter case, the graphics engine can either be of the 'Geo-Mod' type [Red-Faction], or the object itself can simply have been specially marked as destructible and have an alternative, 'broken' graphical representation.

The Octree

Those considering implementing these CA methods will have quickly noticed that storing half-meter cells for even a modest-size level consumes a huge amount of memory, and the processing and memory bandwidth requirements become severe. The approach to doing this efficiently is to not store or process cells that are not participating in any interesting activities—notably inert walls and/or air at (standard) ambient temperature and pressure (STP).

An octree is ideally suited to storing this arrangement, specifically a dynamically allocated octree. In any implementation of the octree, remember that the most common operation in a CA is "find the cell next to me," so it makes sense to optimize for this type of operation when implementing the octree. If this request is made and there is no neighboring cell in the octree, it is assumed that the neighboring cell is air at STP. The physical simulations are carried out accordingly; and if they result in the 'missing' cell becoming significantly different from STP, a cell with the new properties is created and inserted in the octree. When an air cell returns to within a certain tolerance of STP, it is deleted from the octree and is no longer processed.

The octree holding CA cells can also be useful as a general-purpose octree. Many games use octrees to optimize collision detection and visibility culling, and there is no reason the octree cannot fulfill both roles and hold objects not directly related to the CA. A fairly easy adaptation to the search algorithms allows the octree to become a 'loose octree' [Ulrich00], which has several other advantages over a conventional octree. This does not change its behavior when dealing with the CA aspect of its behavior, since all CA cells are aligned to regular intervals and have a fixed size.

Practical Physics

The main thing to remember when writing CA physics routines is to keep things simple. It is surprisingly easy to write very simple routines that take major physical shortcuts, yet look perfectly natural to the player. As long as the basics of conservation of mass and energy are retained—which is frequently optional—most of the other code deals with keeping the simulations stable.

The major problem we encountered during implementation was finding good, simple models of various physical features. Most of the standard references deal with the application of Navier-Stokes equations for various materials and implementing them with as little error as possible. This enormously complicates the code, and most of the academic and commercial literature is concerned with these error reductions. For games, what is required is simplicity, not accuracy. Most of the time, finding implementations involved getting only the general feel of the behavior from the literature.

Core Processing Model

Most of the properties simulated by the CA work in similar ways. To illustrate these common methods, here is a very simple fluid simulation that just tries to achieve even

distribution of pressure throughout the available space. Even this simplified model is very useful for air and fluid modeling.

```
for ( neigh = each neighbor cell )
{
    if ( neigh->Material->IsInert() ) continue;
    float DPress = cell->Pressure - neigh->Pressure;
    float Flow = cell->Material->Flow * DPress;
    Flow = clamp ( Flow,
        cell->Pressure / 6.0f,
        -neigh->Pressure / 6.0f );
    cell->NewPressure -= Flow;
    neigh->NewPressure += Flow;
}
```

The `clamp()` operation is performed to prevent `NewPressure` from going negative. The division by six is because there are six neighbor cells. In practice, even more damping might be needed to retain stability and prevent small oscillations, such as waves on the surface of water, from becoming unrealistic oscillations.

Conventionally, once all the cells have been processed in this way, the `NewPressure` values are copied to the `Pressure` values. This double-buffering is necessary, rather than simply writing directly to `Pressure` at the end of the routine. Otherwise, pressure will be transmitted extremely fast (sometimes instantly) in the direction that the cells are updated, and much slower in the reverse directions. This produces obvious asymmetry in heat distribution, water flow, and other processes.

The double visit to each cell can hurt performance considerably, especially as the second visit is simply a copy, and will be limited by memory bandwidth on most modern CPUs. A better method is to store the last turn that a cell was processed. When subsequently processing that cell, the turn number is checked; and if it is earlier than the current turn, the copy is done. Although slightly odd-looking, this is in fact much quicker than scanning the whole array of cells twice. The code becomes:

```
if ( cell->Turn != CurrentTurn )
{
    cell->Turn = CurrentTurn;
    cell->Pressure = cell->NewPressure;
}
for ( neigh = each neighbor cell )
{
    if ( neigh->Material->IsInert() ) continue;
    if ( neigh->Turn != CurrentTurn )
    {
        neigh->Turn = CurrentTurn;
        neigh->Pressure = neigh->NewPressure;
    }
    // same physics code as before
}
```

Air

This simple model works well for uniform redistribution of air pressure. At first glance, this is not something that is frequently modeled in games. But in fact, it is one of the most common effects—explosions and their effects on things. An explosive is simply a lump of material that produces a huge amount of air in a very short time. They can be modeled through the following steps. First, find the nearest CA cell to the center of an exploding grenade. Second, add a large number to the cell's pressure. Third, let the CA propagate the pressure through the world. Damage is done to the surroundings by either high absolute pressures or high pressure differences—in reality, both do different kinds of damage to different objects; but that is usually unnecessary complication for the purposes of a game.

The advantages of this method of modeling over conventional ones is that line-of-sight is handled automatically. Explosions in confined spaces are far more deadly at a certain range than explosions in open spaces because there is less space for the pressure to dissipate. In addition, it shows that pure line-of-sight is not protection enough from explosions—they do go around corners and obstructions to a certain degree.

Because the simulation of the flow of air is qualitatively correct to the human eye, debris and small objects can be carried along with the explosion; you don't have to worry about the illusion being shattered by debris going the wrong way or through solid walls.

Water

Water is only slightly more complex than air. The obvious distinction is that air expands to fill the available space with cells changing pressure to do so, while water stays at the bottom of its container and is incompressible.

In fact, the easiest way to simulate the transmission of pressure through water is to make it slightly compressible. This means pressure can be stored as a slight excess mass of water in the cell, above what the cell's volume should be able to hold. In practice, the amount of compression needed is tiny—allowing just 1% more water per cell per cube height is easily enough. In a static body of water whose cells can normally contain 1.00 liter of water each, the cells at the top will contain 1.00 liter, the ones under them will contain 1.01 liters, the cells under those will contain 1.02 liters, and so on to the bottom. This tiny amount of compression will be completely unnoticeable to the player, but it has enough dynamic range to allow all the usual properties of liquids. For example, the levels of water in two containers joined by a submerged pipe will be the same, even if water is poured into one of them; it will flow through the pipe to the other container.

```
if ( neighbor cell is above this one )
{
    if ( ( cell->Mass < material->MaxMass ) ||
```

```
            ( neigh->Mass < material->MaxMass ) )
        {
            Flow = cell->Mass - material->MaxMass;
        }else{
            Flow = cell->Mass - neigh->Mass
                   - material->MaxCompress;
            Flow *= 0.5f;
        }
    }
    else if ( neighbor cell is below this one )
    {
        if ( ( cell->Mass < material->MaxMass ) ||
             ( neigh->Mass < material->MaxMass ) )
        {
            Flow = material->MaxMass - neigh->Mass;
        }else{
            Flow = cell->Mass - neigh->Mass
         + material->MaxCompress;
            Flow *= 0.5f;
        }
    }
    else    // neighbor is on same level
    {
        Flow = ( cell->Mass - neigh->Mass ) * 0.5f;
    }
```

This `Flow` value is then scaled and clamped according to some measure of the maximum speed that the fluid can flow, allowing some fluids to appear more viscous than others, and to prevent any resulting masses from going negative.

The two cases of code for the water model deal with different situations. The first case is where one of the two cells is not full of water—such as on the surface of a body of water or if the water is splashing or falling (e.g., in a waterfall). Here, the behavior is simple—water flows downward to fill the lower cell of the two to the value `MaxMass`—the mass of water that can be contained by a single cell's volume. In the previous example, the mass is one liter of water.

The second case is where both cells are full of water, or perhaps a bit over-full, such as in the middle of the body of water. Here, the flow acts to try to make sure that the upper cell has exactly `MaxCompress` more water than the lower cell. `MaxCompress` is the amount of 'extra' water that can be fitted in because of compression. In the previous example, it would be the mass of 0.01 liters of water.

Flow

So far the air and water models have ignored a fairly important property of any liquid or gas—its speed of flow. We have simply taken the difference in pressures between two cells and used that to move mass around. This is fine for relatively static environments that we wish to bring to a stable state, such as uniform air pressure or water finding its level. Many games will only use these simple properties for all the gameplay and realism they need.

However, what happens in real life is that water and air have momentum (which equals flow times mass), and the difference in pressure only influences the flow between cells; it does not rigidly set it. Storing momentum or flow is important when modeling waves, flowing rivers, and air currents. Although rivers can flow in models without momentum, they have a very visible slope of at least 10°, which looks very bizarre.

To model momentum or speed of flow during each processing step, the difference in masses determines the pressure gradient, as before. However, instead of changing the masses of the cells directly, the pressure gradient only alters the flow between the cells. The flow then changes the masses in the cells. The code is slightly more complex because flow is a three-dimensional vector and not a scalar like mass.

There are two possible ways to think about flow. The first is to think of a flow vector as being the flow through the center of the cell. This is possibly the most intuitive model—the flow and the mass of the cell are both measured at its center. However, in this case, the flow is affected by the pressure differential between the two neighboring cells, which in turn determines how mass flows from one neighboring cell to the other. Note the slightly odd result that, for a particular cell, the flow stored in it is not affected by the mass in the cell itself, but only by its neighbors. Nor does it change the mass of the cell, but only the mass of its neighbors' cells. This is a slightly surprising result; and in some cases, this can lead to odd behavior.

It is more useful to think of each component of the flow vector as being the flow between two adjacent nodes—from the 'current' node to the node in the positive relevant direction. Thus, the flow vector **F** stored at cell (x, y, z) is interpreted as meaning that $\mathbf{F}_x$ is the flow from cell (x, y, z) to cell $(x + 1, y, z)$; $\mathbf{F}_y$ is the flow from cell (x, y, z) to cell $(x, y + 1, z)$; and, similarly, for $\mathbf{F}_z$. The 'meaning' of the vector **F** is now not as intuitive, but the physical model does seem more sensible. In practice, this is the most common model; but either model can be used for simulation with appropriate adjustment of the various constants.

The most important step in this model is to carefully control oscillations. Not only does this model allow waves, but it also tends to encourage them to build up, and sufficient damping must be applied to the flow by introducing a simple friction coefficient. Otherwise, waves can build up higher and higher instead of dying down, and the liquid or gas starts to do very odd things indeed.

It is worth mentioning that, although one of the most common applications of flow is in rivers, in most 'human-size' games, large bodies of water, such as lakes and rivers, are frequently far too large to participate in gameplay. Their behavior will stay fairly constant whatever the player does; and if they do change, they will do so in highly constrained ways. They do not usually require the flexibility of a CA and are often far better modeled and rendered in more-conventional ways. We can use pre-animated meshes, collision models, and scripted events. However, there are many other genres that operate on larger scales and will want to properly simulate rivers with a CA.

Heat

Transmitting heat through the environment, whether from burning objects or from other sources, happens through three separate mechanisms: conduction, convection, and radiation.

Conduction

Conduction is the simplest mechanism to simulate. Neighboring cells pass heat energy between each other so that eventually they reach the same temperature. This is complicated because different materials are heated by different amounts by the same energy—called the specific heat capacity (SHC), which is usually measured in J/kg°C. If a hot cell made of water (high SHC and hard to heat up) is next to a colder cell made of the same mass of iron (low SHC), equilibrium will be reached at somewhere very close to the original temperature of the water, not at the average of the two temperatures. This is because when a given amount of energy is transferred from the water to the iron, the water's temperature drops far less than the iron's temperature rises.

Note that the above example is true for the same *mass* of each substance. However, iron has a far greater density than water; and therefore, for the same *volume*, they have very similar heat capacities.

```
// Find current heat capacities.
float HCCell = cell->material->SHC * cell->Mass;
float HCNeigh = neigh->material->SHC * neigh->Mass;
float EnergyFlow = neigh->Temp - cell->Temp;
// Convert from heat to energy
if ( EnergyFlow > 0.0f )
    EnergyFlow *= HCNeigh;
else
    EnergyFlow *= HCCell;
// A constant according to cell update speed.
// Usually found by trial and error.
EnergyFlow *= ConstantEnergyFlowFactor;
neigh->Temp -= EnergyFlow / HCNeigh;
cell->Temp += EnergyFlow / HCCell;
// Detect and kill oscillations.
if (((EnergyFlow>0.0f)&&(neigh->Temp<cell->Temp))||
    ((EnergyFlow<=0.0f)&&(neigh->Temp>cell->Temp)))
{
    float TotalEnergy = HCCell * cell->Temp +
                        HCNeigh * neigh->Temp;
    float AverageTemp = TotalEnergy /
                        ( HCCell + HCNeigh );
    cell->Temp = AverageTemp;
                        neigh->Temp = AverageTemp;
}
```

The code at the end is necessary if two materials with very different SHCs are side by side. In this case, the temperatures of the two can oscillate violently and can grow

out of control. The physically correct solution is to integrate the transfer of heat over time. However, this approach simply finds the weighted average temperature, which is the temperature that the system would reach eventually. It is less accurate, but looks perfectly natural and is quite a bit quicker to execute. Importantly, it obeys the law of conservation of energy, so any artifacts are purely temporary. The longer-term state is the same as a more realistic simulation.

Convection

Convection is the phenomenon of heat rising. Hot areas of fluid, such as air or water, are less dense than cold areas, and thus try to rise. This can be simulated by incorporating temperature into the model of water or air. If a flow model is being used, the flow will be influenced by the relative temperatures of cells as well as their relative pressures. Otherwise, convection does not work very well, though its effects can be faked as described in the section on fire.

Radiation

Hot things glow. They emit light at various wavelengths, which travels in straight lines, hits other surfaces, and in turn heats them up. This effect is very important physically, but unfortunately it is also extremely expensive to model. Each source of heat must effectively shoot many rays out from itself and heat up whatever they hit.

Radiative heat modeling is very similar to the radiosity modeling that is used when creating lightmaps for many current games. Both are extremely expensive to model in runtime, even crudely, though there are some cunning methods that use a heavy amount of approximation to improve the speed of radiative heat modeling. Even with these algorithms, modeling even a fraction of the radiative heat seems like a prohibitive amount of work for a game. These algorithms are also extremely complex and do not involve the standard cell-to-cell interactions that model all the other physical properties mentioned. For both these reasons, we won't discuss them here.

Fire

The physics of burning materials is frequently extremely complex. There are multiple parts that burn at different rates and heats, and there are also different phases of material involved in the process.

To perform the calculations in real-time, the material models used during the process need to be trimmed down to their minimum; and for each material, an appropriate model must be chosen that emphasizes the main characteristic.

Of the many models considered, the one that finally seems to give the best results for the least amount of effort is a quadratic approximation of an exponential graph. This graph shows how much heat energy is released per unit of time when a substance burns at a certain temperature. There is a maximum amount of energy that can be released, no matter how hot the fire gets. But even at relatively cool temperatures, a lot of heat is

released. This explains why open fires tend to start small, rapidly grow to a certain size, and then not grow any bigger, yet burn for a long time, despite ample availability of fuel. They are simply not generating enough heat energy to compensate for the heat lost to the environment (which is directly proportional to the temperature).

```
float Temp = cell->Temp - material->Flashpoint;
// Damage the cell.
CellDamage = Temp * material->BurnRate;
float Burn;
// Convert to actual burning value.
if ( Temp > material->MaxBurn * 2 )
    Burn = material->MaxBurn;
else
    Burn = ( 1.0f - ( 0.25f * Temp / material->MaxBurn ) ) * Temp;
ASSERT ( Burn <= material->MaxBurn );
ASSERT ( Burn >= 0.0f );
// And heat the cell up from the burning.
cell->Temp += Burn * material->BurnTemp;
```

Note that the damage done to a cell is proportional to its actual temperature, not how much heat is generated by burning. This allows materials that burn at low temperatures to nevertheless be far more severely damaged if exposed to high temperatures. By varying the factors `MaxBurn` and `BurnTemp`, burning anything can be simulated—paper, wood, oil, gunpowder, or high explosives.

Of course, one of the major aspects of fire is that it is hot, and thus it relies heavily on the modeling of heat flow by the three methods discussed previously. In real-life fires, convection and radiation are incredibly important to their behavior. Convection makes fires spread vertically far easier than spreading horizontally, such as across floors, and leads to distinctive 'walls of fire' in burning buildings. Radiation concentrates fire in corners of rooms, causing fire to spread up the corners of the room first.

Sadly, radiative heat, as mentioned above, is extremely hard to model, and convection, although slightly more straightforward, requires large numbers of air cells around the source of the fire to be modeled and updated, which is expensive. It would be far better to find some hacks that simulate some of these features without incurring the considerable expense involved.

A hack for convection effects is simply to make conduction of heat far easier in an upward direction. In real life, a section of burning wall heats the air beside it, which rises and heats the section of wall higher up. This makes it far easier for the flames to spread upward. Using this hack, conduction of heat is made artificially asymmetrical. In the model presented above, a single factor—`ConstantEnergyFlowFactor`—was used for heat conduction for all six neighbors of a cell. Instead of this, a higher figure is used when conducting heat upward and a lower figure when conducting heat downward.

A hack for radiation is more difficult, but it is possible that some precomputation could be done using the same techniques as radiosity to decide which parts of a room

would be more susceptible to fire because of the feedback effects of radiative heat. One possibility is computing the hemispherical occlusion term [Hemispherical01] and using that to boost the heat generated by fire—generally around edges and corners.

A factor that might not be immediately obvious is that these hacks are far more controllable than any realistic solution. Convection in real life is a notoriously chaotic system, and small changes in conditions can cause it to adopt very different patterns of flow. This can make designing gameplay around the effect very tricky indeed. What game and level designers usually require is a high degree of control and predictability to carefully create exciting set-pieces for the player to experience. The hacks presented above are far more predictable and linear in their behavior, which is usually a far more desirable quality in a game than absolute realism.

Dynamic Update Rates

The natures of some of the physical properties being simulated here require high update rates to maintain realism. The flow of any property from one cell to another can only proceed at a maximum speed of one cell per update cycle. Fire might spread quickly—at meters per second or faster. Water spilling from a container might move even faster—at tens of meters per second. Explosions require extremely high update rates—real-life explosion shock waves spread at the speed of sound, roughly 340 m/s.

Simulating all of the above implies that update rates of 680 cycles per second could be required. This is an awesome speed, and it seems unlikely that any current platform can sustain these sorts of update rates for a decent-size game world.

As with the optimization of not storing or processing cells at STP, it is possible to use the octree to reduce the update rates for cells that do not require fast updates to maintain realism.

When a cell is processed, it decides how fast it needs to be updated to maintain a good simulation, based on its current state. Cells involved in explosions require high update rates; cells holding flowing water, burning objects, or high heat need medium update rates; cells with fairly static water require lower update rates; and cells that hold scenery at ambient temperature require no processing at all until disturbed.

This speed of processing is then stored in the cell and is also passed up the octree hierarchy, each level being marked so that it is processed at the highest update rate of any of its children. This then allows the update routine to start at the top node of the octree and recurse down the tree. At each level, it decides if the current node would require processing of the all the child nodes.

One point to note is that this system only works if the update rates are quantized to powers of two. For example, if a child node needs to be updated every third turn, but the parent node is marked as being updated every second turn, every sixth turn the child node needs updating, but the parent does not. Because of the traversal algorithm's early-out path, the child does not get updated this time, and in the end only

gets updated every sixth turn. Quantizing update rates to powers of two solves this problem and also allows some slight extra storage efficiency.

An obvious consequence of this variable update rate is that the physics routines need to be able to handle variable update rates as well. So far, all the code has assumed that it will be run at a set speed, and that the physical constants will be adjusted to give good results for that speed. With variable update rates, the physically correct behavior is to integrate the various equations over the given period. However, one of the purposes of the variable update rate is to choose an update rate that ensures the cells have a fairly constant behavior over the update interval.

This assumption makes integration simple. We just multiply the given behavior flow or rate by the time period since the last update. This slight extra complication is more than offset by the savings in processing time and memory bandwidth because of the huge reduction in the number of cells updated per second.

In many cases where the mathematics are simple, it might be more efficient to actually perform the integration. The extra accuracy of the simulation will then allow the use of an even lower update rate, further improving speed overall.

An unexpected artifact of using a variable update rate can occur when neighboring cells have very different update rates. Since one cell is being updated much more frequently than its neighbor, it can change rapidly before the other cell has time to react. The solution is to limit the maximum difference in update rates of adjacent cells. Every time a cell is processed, as well as exchanging temperature, heat, and similar information with neighboring cells, it also ensures that those cells are being updated at least a quarter as fast as itself. We found this factor purely by experimentation. Using a factor of two causes too many cells to have their update rates raised pointlessly by nearby events, when in fact nothing exciting is happening to them. Using a factor of eight or more allows more possible artifacts, but does not reduce the processing load appreciably. As with the many arbitrary factors that are found by pure experimentation, others should experiment to find what works best.

Conclusion

The use of CA methods allows the simulation of a wide range of real-world effects and situations rarely seen in games today. They allow the player to interact with them fully, flexibly, and logically, without the limitations of prescripting. This opens the way for more inventive puzzles, more lateral thinking by the player, more freedom to experiment, more realistic rendering, and overall better immersion in the game world as a real place, rather than as a collection of polygonal entities.

References

[CFD] There are innumerable references to the field of computational fluid dynamics. Unfortunately, most assume graduate physics knowledge or an extremely firm

grasp of 3D calculus. One of the more comprehensible of these can be found at http://www.efunda.com/formulae/fluids/overview.cfm.

[Conway] http://www.dmoz.org/Computers/Artificial_Life/Cellular_Automata/Conway's_Game_of_Life/.

[Hemispherical01] "Advanced Shading and Lighting," presentation at Meltown 2001: pp. 22–35. Available online at *http://www.microsoft.com/mscorp/corpevents/meltdown2001/ppt/DXGLighting.ppt.*

[LorensenCline87] Lorensen, W. E. and Cline, H. E., "Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm," *Computer Graphics Proceedings* (SIGGRAPH 1987), Vol. 21, No. 4: pp. 163–169.

[RedFaction] *Red Faction,* developed by Volition, Inc. Published by THQ, Inc., 2000.

[Ulrich00] Ulrich, Thatcher, "Loose Octrees," *Game Programming Gems,* Charles River Media, Inc., 2000.

[XCom] *X-Com: UFO Defense* (U.S.) or *X-Com: Enemy Unknown* (U.K.), Microprose, 1994, http://www.codogames.com/UFOUnknownDefense.htm.