☞ MicroCosm™ ☜

a proposal for a new form of interactive entertainment
by
Chip Morningstar

Lucasfilm Ltd. Games Division
21 September 2020

Introduction

MicroCosm™ will be a multi-user interactive simulation of an imaginary alternate reality. Distributed processing techniques will enable a large number of people to become vicarious inhabitants of this other place. Their home computers, tapping into our network using low-cost modems and the household telephone, will be their windows into it.

MicroCosm™ is, in some sense, a role-playing game. Each player takes the part of a character in the vast, on-going drama. Running on each player's home computer is a piece of software that acts as that player's window into the world. This program presents to the player the sights and sounds that his or her character would be experiencing in the situations that the character encounters, together with interactive controls to direct the character's actions and movements within the simulation. The home computer connects the character to the goings on in the rest of the world by means of data communications over the telephone. This connection is routed through a central network host to the other players' home computers and to a central coordinating process that resides within the network host itself.

The Primal World

The primal world is a broad, featureless plain. We'll call it the *Plain* with a capital '*P*'. Aside from being a plain (plane), the topological details haven't been decided yet. The Plain could be the surface of a sphere, or of a torus. It could be an infinite plane extending in all directions, or it could be bounded in some way. Such a boundary could be an impenetrable wall, an unsurvivable desert, or a sheer precipice (the edge of the world!). Or, we could use some combination of these. For example, the side surface of an infinite half-cylinder gives an axis along which you can travel forever going one way or reach the edge of the world going the other, and an orthogonal axis along which you can circumnavigate the world and wind up back where you started!

We said that the plain is featureless. This is to keep things simple for the network host since it has many players that it has to keep track of at once. In fact, the plain isn't *entirely* featureless. It has some sort of texture to keep it from being completely blank. This might be a gridwork of some kind, or a checkerboard pattern`a la **ballblazer**, or some type of stochastic patterning such as intermittent streaking or fractal dirt. The texture performs a couple of important jobs for the player. It provides something for the player's eyes to focus on for depth and motion cues and gives a frame of reference for judging direction and orientation. This texturing information is not generated by the network host, however. The player's home computer simply knows the player's position and orientation and can generate the texture locally. Different players need not have the same image of the primal world -- there is no one ''right'' texture pattern. This enables us to support different models of home computers without worrying overly much about keeping the appearance of the world constant across them. For stylistic purposes, however, we should decide whether we prefer a geometric pattern (e.g., a checkerboard) or a stochastic one (e.g., textured fractal dirt).

Basic Concepts

Another thing that keeps the Plain from being a total blank are the things found there. There are three important concepts to reckon with in this regard: *objects*, *avatars* and *turf*.

Objects are items that can be found in the world: independent, inanimate things, such as rocks, trees, cars, stereos, light-bulbs, boxes, weapons, chairs, and so on. Objects all share a common set of basic properties (e.g., location). In addition, a few of them have special properties that are handled by special software running on the network host. Objects have some important functions that will be discussed below. Their main job, however, is simply to give the players something to manipulate.

Avatars are the players' bodies in the fantasy. The players each have an avatar through which they interact with the world, with the objects in it and with the other players' avatars. Avatars are little individual homunculi that the players can control (the controls themselves are discussed below). Avatars have various traits in the fashion of characters in traditional fantasy role playing games (e.g., strength, height, appearance, etc.). Some of these traits are determined stochastically for the players by the system, and some are personally chosen by the players themselves.

Turf refers to pieces of the Plain that ‘‘belong’’ to individual players. The notion of turf is what makes the world work in the distributed environment. The idea is that a player can ‘‘own’’ a piece of the Plain, in the sense that that piece is marked off in the network host's central database as belonging to the player. The interior of a player's turf is simulated by software running on his own home computer, thus, unlike the bland, open Plain, it can have as much graphic and topological complexity as the owner's computer can handle. Any other player whose avatar enters that particular region of the Plain is, in essence, connecting the avatar-simulator in his own machine to the world-simulator in the first player's machine.

Player Interaction

The player interacts with the world by controlling the actions of an avatar: a software homunculus who is ‘‘really there’’. We are trying to project the player into the world. To do this effectively the range of actions available to a player's avatar should be broad. This poses a design problem since the available input options for the various anticipated target machines are not broad at all: we have a two-dimensional pointing device (either a mouse or a joystick) with a push-button on it and we have an ASCII keyboard. The single pointing device does not provide enough degrees of freedom to control all the dimensions of behavior that we wish to manipulate — motion, posture, orientation, arm and hand actions, and so on. The unfortunate but inescapable conclusion to be drawn from this is that the player interface will have to be based on the keyboard.

By our reckoning there are three qualitatively different kinds of inputs that the interface needs: 1) commands to control the immediate actions of an avatar, 2) verbal communications between one player's avatar and another's, and 3) commands to establish semi-autonomous behavior for an avatar. The latter is for situations where quick reaction is required or where the player will be absent from his machine for a brief period (e.g., to take a pee break) but wishes his avatar to continue functioning in the world during that time. In a sense these three categories correspond to *actions*, *words*, and *thoughts* respectively.

One possible interface would use three ‘‘windows’’ on the screen. There would be one window for each class of input. The interface would presumably provide a quick and easy means of shifting attention from one window to another. Each window would recognize an input ‘‘language’’ most appropriate for the sort of activity that takes place in it. For example, the *actions* window might recognize a set of quick, single-keystroke commands that direct the actions of the avatar. The*wor ds* window, on the other hand, would simply accept typed text and transmit it to any other players whose avatars are within speaking distance. The *thoughts* window might be something like a miniature programming language editor in which the player could compose macro-like packages of action commands. A fourth window would also be required, to provide output. This would be a graphic display showing the player what his avatar is seeing, and so we'll call it the *sight* window. It is important to note, however, that this interface design is really just speculation to illustrate some basic needs. In all hopes the final design will be more sublime.

System Design Goals

The system is divided physically into two parts: the centralized network host (which we'll call the *host*) on the one hand and all the various distributed home computers (which we'll call the *remotes* or the *player systems*) on the other. An obvious but important distinction between them is that there is only one host, whereas there are many, many remotes. We assume that each remote corresponds to one player. The system grows as players are added to it, but each new player brings a new CPU into the system. Thus, the bottleneck is the host. To keep the host system CPU requirements from blowing up in our faces we must minimize the host computational load as much as we can and make the remotes do most of the work. The first rule of system design must therefore be *Spread The Load*.

It is theoretically possible to design a completely distributed system, so that the load is entirely on the remotes and the host acts as a trivial communications switch, if indeed there is a host at all. However, because of the need to maintain the integrity of the overall world, it is not possible to reliably distribute certain functions and capabilities. This is because no individual player system can be considered entirely reliable nor can we afford to trust the users with certain powers. There is no way to guarantee that the software running on any given remote is our own software unmodified. In particular there is no way to assure that a remote won't "cheat" on its owner's behalf or introduce irregularities into the world. We have to be sure that players will not be able to damage other players by dint of clever hackery. The only way to do this is to build interlocks and handshaking protocols into the host that guarantee that the only person a player can mess up is himself. We can't prevent a player from destroying his own piece of the world, but then we can't keep him from other self-destructive acts either (such as smashing his computer, jumping off the Golden Gate bridge, or failing to pay his system connect-time bill). The second rule of system design is thus *Keep It Secure*.

The need to support a large number of remotes (so that we need to keep the load down) while at the same time reserving certain activities to the host alone (which tends to push the load up) implies a third important rule of system design: *Keep It Simple*.

The Design

Each player has an *account* on the host system. Each account corresponds to a single imaginary inhabitant of this imaginary world. If a player wishes to play multiple roles he must have multiple accounts, which we will not try to prevent. We will sometimes speak of the player, his account, the name of his account, and the persona of the avatar he is controlling as if they were all the same thing. Even though Korzybski reminds us that this is not true, it will occasionally simplify discussion.

Things in the world have *properties*. Properties are any of a number of abstract attributes that an object, avatar, or piece of the Plain may possess. Certain properties in particular are very important to the function of the system.

One important property is *ownership*. Each object, avatar, and region of the Plain has an *owner*. The owner is usually a player, though some things are owned by special system accounts. Ownership of a thing in this world is a subtly different notion from ownership of a thing in the "real" world. In this world, generally speaking, only the owner of a thing is entitled to manipulate it in any way. There*ar e* exceptions to this broad generalization, but they are not important just yet.

Another important property that all things have is *location*. Each thing has to*be* some where. Location is important because of the notion of turf. Whether an object or an avatar is located on its owner's turf, on some other player's turf, or on nobody's turf (i.e., the system's turf) together with the identity of its owner determines what combination of computers is simulating that thing's existence. Location also determines whether one thing may act upon another. However, the geometry and distance metric for the world, like the topology, have not yet been determined.

Introduction To Distributed Processing — Movement

Avatars, being little simulated people, can move around on the Plain by walking (under the control of the players that own them, of course). They may also move around in *vehicles*, which are a special kind of object, but that is not important right now. An exchange between the host and the remotes handles movement in the following manner:

The avatar has a location (which is simply one of its properties as previously discussed) which is known to both the host and to its owner's remote. This is location is in one of three kinds of places: somewhere on the player's own turf, somewhere on some other player's turf, or somewhere on the Plain that is unowned. In all cases, to execute the motion the player's computer simply informs the host of the move by providing the location that has been moved to. The host checks to see if this is a legal move (e.g., that it doesn't move the avatar farther than it would be able to walk were it a real creature). If the move is not legal, the host responds with an appropriate error message. Otherwise, the host updates its database to reflect the avatar's new location, and informs various ‘‘interested parties’’: remotes with a need to know about the change in location. This includes the owner of the avatar, who gets a confirmation of his move, any other players whose avatars are within ‘‘seeing distance’’ of either the new or the old locations, and possibly the remote of the player on, onto, or off of whose turf the avatar has moved, if that is what has happened. These remotes are then responsible for taking any necessary actions to inform their owners of the change in state. If the move takes place on a player's own turf, no further action is required. If the move takes place on the open Plain, the host also informs the player of any objects, avatars, or turf that have come into sight as a result of the move. If the move ends on some other player's turf, that other player's remote *should* (but is not obligated to) respond with this information.

Because we cannot really trust any given remote, and because we are trying to keep the load off the host, the concept of turf introduces modifications to the above procedure. As far as the host is concerned, all locations on a given player's turf are considered interchangeable. In other words, if an avatar is *somewhere* on its own turf, it can be *anywhere* on its own turf. If the remote simulating the turf tells the host of a move to another location within the bounds of that turf, the host simply accepts it, no questions asked, no matter how large a move it might be (the same applies for the location of objects, which we will discuss shortly). This allows any amount of activity to take place on a player's turf ‘‘off-line’’. The remote need only inform the host of the end result. A consequence of this is that an avatar can ‘‘teleport’’ instantaneously from one part of its demesne to another. This is a bit weird, but that's just the way the world works.

Motion on another player's turf obeys similar laws, except that it requires the active cooperation of the foreign player's remote. If the foreign player is not logged in to the host at the time, or if he merely doesn't want to interact with you, his turf is simply impassable. As long as activity is taking place entirely on the other player's turf, the host can act simply as a communications switcher between the two remotes. It is up to the two remotes to decide whether or not they ‘‘believe’’ each other about what is happening. A player can always take his avatar off someone else's turf by informing the host of a move to adjacent territory. Without assistance from special objects (in transactions mediated by the host), a player cannot permanently alter the state of another player's avatar even if that avatar is on his own turf (and therefore ‘‘inside’’ his own computer).

Imagery

What a player sees as all this is happening is up to the user interface software on his remote. The appearance of a player's own turf can be anything that his machine cares to display. We will provide default environments that the players can then customize to their own liking. What the form of these environments will be and in precisely what form the user interface we build will display them has not yet been determined. There are many possibilities. One such possibility is to show a player's turf as a different pattern on the Plain. This is the dullest alternative. More interesting displays might include three-bedroom ranch houses with custom floorplans, abstract sculptures, sets from fifties movie musicals starring Fred Astaire, tropical gardens, sleazy spaceport bars, or retail software chain outlets. The primary goal is to give enough flexibility to allow the creative player to express himself while providing enough structure so that the uncreative player can have an interesting turf without too much work.

Similarly, the user interface can provide imagery for the other player's avatars, for the other player's turf, and for the various objects that abound. We will, however, adopt the convention of allowing the owner of a thing to specify an appearance for it. Thus, when you meet another player's avatar, the host gives you images to display that the other player has provided. A thing's appearance is another one of its properties. This property can be dictated — usually — by the thing's owner. The appearance of another player's turf can be provided by an ongoing real-time dialogue between the other player's remote and your own. Alternatively, static images may be used. In particular, you can provide the host with images to represent your turf when you (or rather, your remote) are not around to represent it yourself. Thus, if you approach another player's turf while that player is not logged in, you may be confronted with the scene of a locked house, or brick wall, or a closed hatch in the ground, or a sign saying ‘‘**BUZZ OFF!**’’.

Objects And Actions

Just moving around on the Plain is not interesting enough. The players need something to do. The way we deal with this is to provide the *ability* to do things and then let the players come up with things to do for themselves. Essential to making this happen are objects. Objects have two major functions. First, they provide physical and visual features for the landscape. Second, they provide active capabilities that avatars alone do not possess, acting as tokens to mediate transactions between players that the player's remotes by themselves cannot be trusted to handle securely.

Objects add physical and visual features to the landscape by their mere presence. Any time an avatar comes within visual range of an object, the host informs the player's remote. The object can then be used in whatever visual display the user interface presents to the player. Objects can have the property of occupying space, and thereby render portions of the Plain impassable by obstruction.

Objects can act as tokens to mediate transactions between players. This is needed because of security considerations: no remote can be absolutely trusted not to cheat. By adding a level of indirection to transactions that necessitates an interaction with the host database, players can reduce the risks of any transaction to those risks that are inherent in the rules of the game, rather than the much broader risks entailed in trusting another person's computer implicitly. In other words, the burden of trust is shifted from the other player to the host. Presumably the host can be trusted, otherwise the world can't work at all.

Objects give avatars the ability to do things that avatars alone cannot do. These objects act as tools to manipulate the state of the world. For example, as explained above, a player cannot directly alter the state of another player's avatar in any permanent way, because he (or rather, his computer) cannot be trusted to always do so within the rules. However, if an avatar possesses a *weapon* object, this changes. A weapon object has a property that enables its carrier to forcibly subtract from the *health* property of other avatars. Like movement, this action is initiated simply by having the remote inform the host that its player is exercising the capability (along with the location of the target, of course). Again as with movement, the host does the appropriate consistency checks (is the target within range? is the gun loaded?), makes the appropriate modifications to the central database, and informs all relevant players of what has happened. Another, less gruesome, example is a *vehicle*. A vehicle is an object that possesses two special properties: the *container* property and the *mobility* property. An object with the container property can contain other objects (up to some limit of size or mass). An object with the mobility property can move around under the control of an avatar inside it, just like an avatar can walk around on its own. An avatar operates a vehicle by loading himself and any other desired objects into the vehicle (the loading itself being a form of movement transaction) and then moving normally (except with an expanded range, speed and/or carrying capacity).

System Architecture

Figure 1 illustrates the underlying system architecture. As discussed above, the system is broken into two parts: the host system and the player's system (the remote). These two parts are connected by a communications channel of some sort (presumably some combination of telephone and packet-switching network linkup). Both ends have a *Communications Channel Controller* that handles the exchange of messages with the other system. The systems communicate using a message-based request protocol that is, as yet, undefined.

The player's system contains an *Internal Modeler* that maintains the player's state in the world. The *Input Handler* accepts commands and control information from the player himself and hands directives to the Internal Modeler. The Internal Modeler attempts to execute these directives directly if it can, by simply updating its notion of the internal state of the local world. If it must, the Modeler sends requests to the host via the Communications Channel Controller. In any case, it passes appropriate commands to the *Graphics, Animation & Sound* module, which updates the visual display appropriately. The Internal Modeler also has access to a *Database Cache* which it uses to minimize the need for transactions with the host that simply request static information (such as the imagery to display a particular object). Directives to the Internal Modeler can also arrive asynchronously from the host over the communications line. These are notifications of actions taken by other players that affect this player, such as movement into the player's visual range or onto the player's turf. Some of these external requests merely require that the player's system update its world model, while others require a response to the host or to another player.

The host system contains a *Message Switcher* that routes requests and messages between one player and another and between a player and the more active part of the host, the *Transaction Monitor*. The Transaction Monitor is responsible for serializing all the various requests arriving from the many remote systems. It decomposes these requests into primitive database transactions which it then hands to the *Database Controller* for execution. The*Database Contr oller* adjusts the database accordingly and passes a response back to the Transaction Monitor which in turn responds to the requesting remote. The Database Controller also detects changes in the state of the world that require notification of non-requesting players. Information about these is sent to the *Asynchronous Notifier* which identifies the relevant parties and requests the Transaction Monitor to respond to them as well.

A Plan Of Attack

The development effort for this system splits into the following tasks. Approximate labor requirements for each task (in ‘‘man-months’’) are shown after each description*. In no particular order:

[1] Design and specify message and communications link protocols. *(.5 mm. arch.)*

[2] Design and specify object set and object properties. *(.5 mm. arch.)*

[3] Design and specify object database. *(.5 mm. arch.)*

[4] Design and specify user interface. *(.5 mm. arch.)*

[5] Design and specify multi-player transaction processor. *(.5 mm. arch.)*

[6] Build host Communications Channel Controller. *(1 mm. host prog.)*

[7] Build remote Communications Channel Controller. *(1 mm. micro prog.)*

[8] Build host Database Controller and skeleton Database. *(2 mm. host prog.)*

[9] Build host multi-player transaction handling modules (Transaction Monitor, Asynchronous Notifier and Message Switcher). *(4 mm. host prog.)*

[10] Build player system Input Handler. *(1 mm. micro prog.)*

[11] Build player system Graphics, Animation & Sound. *(3 mm. micro prog.)*

[12] Build player system Internal Modeler and Database Cache. *(3 mm. micro prog.)*

[13] Create objects and imagery. *(1 mm. prog., 5 mm. art.)*

[14] Integrate and test. *(ongoing)*

[15] Design and document host system administrative and operational procedures. *(.5 mm. arch.)*

[16] Set up operations organization. *(1 mm. admin.)*

[17] Turn it on and let the public at it. *(0 mm.!)*

Many of these tasks are interdependent. For example, the set of object properties, the structure of the database, and nature of the message protocol all influence each other. The usual back-and-forth cycle between design and implementation will also apply. Integration and test will be ongoing rather than discrete activities. Also, implementation work for player systems will, of course, have to be duplicated on each type of target machine (e.g., Commodore 64 and Amiga). Host system development work will have to build on top of the architecture provided by the organization providing the host system.

Assuming two target machines for the player system (thus doubling the amount of player system programming required), this effort will require just under three ‘‘man-years’’ of effort. This load is spread over four people for one year: a full-time project leader/system architect/programmer, another full-time programmer, a half-time programmer, and a half-time artist/animator.

In addition to the above personnel, this project will require use of the Lucasfilm in-house development systems for the player system target machines. We will need access to the host system and such development tools for it as are available. Also, information and assistance from the operators of the host system is needed so that we can familiarize ourselves with the host’s peculiarities and with the baseline

* * *arch.* = system architect, *host prog.* = host system programmer, *micro prog.* = player system programmer, *prog.* = miscellaneous programmer, *art.* = artist/animator, *admin.* = administrator. These are functional descriptions, not necessarily distinct individuals.

network that we are building on top of. Some travel between San Rafael and the host system site may be required in this regard. Finally, there will of course be the usual overhead costs for changing light bulbs, cleaning the bathrooms, watering the lawn, answering the phone, and so on.

Conclusion

Hey, what can you say? We can do it and it’s gonna be fun!