How to create an RTS(real-time strategy)

Audience:

This document is designed for DarkBasicPro programmers that have at least a basic understanding of programming logic and is familiar with the language. If you haven't created anything more complex than Pong, then this may be too involved for your skill at this time.

Goal:

To create a basic RTS engine which includes character selection, moving, attacking, building construction(house, barracks, etc...), and character management. A future article entitled "Polishing an RTS game" is planned and is meant to take these concepts into greater depth.

Begin: Choosing your type of landscape

The first step is to decide how you will design the landscape. The reason being this is because there are different ways to implement the mouse interface that you'll use to select spots in 3D on the landscape. You could use a matrix, but to have accurate 3D mouse positioning you will be limited to a completely flat ground. If that works for you, then use it. It'll keep things more simple later on. However, if you want a rough terrain with some hills and valleys, you'll need to use an object. You could use the new terrain commands recently added to DBP or use the memblock matrix code that myself and a few others(Van-B,Kevil) have written. By using an object, 3D mouse coordinates are easily attained by using the **pick object** and **get pick vector** commands. For those that wish to stick with the standard flat matrix, here is the code used to translate a 2D mouse position into a 3D coordinate. The author of this code snippet is unknown to me.

These variables must be declared at the beginning of your program.

```
screenwidth2# = screen width() / 2.0
screenheight2# = screen height() / 2.0
scalefactor# = screen height() / 1.2
```

The **offsetx#, offsety#, offsetz#** variables is position at which the camera is pointing to.

```
_Click_Matrix_Position:
  mouseposx#=mousex() - screenwidth2#
  mouseposy#=screenheight2# - mousey()

  dist#=sqrt((camera position x()-offsetx#)^2+(camera position y()-offsety#)^2+(camera position z()-offsetz#)^2)

  if mouseposy#<>0
    vectorang#=atanfull(scalefactor#,mouseposy#)
  else
    vectorang#=90.0
  endif

  if vectorang#+camera angle x()>90.965
    ratio#=mouseposy#/(sin((vectorang#+camera angle x() )-90.0))
    cursorposy#=ratio#*sin(180.0-vectorang#)
  else
    cursorposy#=1000000.0
  endif

  hyplength#=scalefactor#/sin(vectorang#)
  cursorposx#=(((ratio#*sin(90.0-camera angle x() ))/hyplength#)+1.0)*mouseposx#

  if mouseposx#<>0
    angtotal#=wrapvalue(camera angle y()-atanfull(mouseposy#,mouseposx#))
  else
    if mouseposy#<0
      angtotal#=wrapvalue(camera angle y()+90.0)
    else
      angtotal#=wrapvalue(camera angle y()-90.0)
    endif
  endif

  cameraang#=wrapvalue(360.0-camera angle y() )

  cosang#=cos(cameraang#)
  sinang#=sin(cameraang#)

  movex# = (dist#/scalefactor#)*((cosang#*cursorposx#)+((-1*sinang#)*cursorposy#))
  movez# = (dist#/scalefactor#)*((sinang#*cursorposx#)+(cosang#*cursorposy#))

  rem 3D position stored into these variables
  movex#=movex#+offsetx#
  movez#=movez#+offsetz#
  movey#=0

RETURN
```

So to use the code properly, you must know where the camera is pointing. Easily done, just pick a spot on your matrix to point the camera at so you have the desired angle of view that you want. Then whenever you call this routine, the 3D position of your mouse cursor will be stored into the **movex#** and **movez#** variables.

Now for those who wish to use an object for their terrain.

```
rem object number of your terrain
object = 1
null = pick object(mousex(), mousey(), object, object)
targetX# = camera position x() + get pick vector x()
targetY# = camera position y() + get pick vector y()
targetZ# = camera position z() + get pick vector z()
```

That's it. Much easier isn't it. So why would anyone want to use a flat matrix with less flexibility when it takes more code to use? The reason I said a flat matrix is easier won't be fully explained until later, but just think about position buildings on your terrain. If you know it's all flat, then how you place the object is quite simple. But think about placing a house or building on a random matrix. A bit more complex task, if you want it to look right. From now on in code, I will only use this function name, **get3DMouse()**, and these variables, **targetX#, targetY#, targetZ#**, when referring to either of the above 3D mouse selection routines.

Step 2: Move your character

Great, we can now get a 3D position from our mouse, but how do we make our character walk to the targeted spot? This is easier than some people might think. Here, see for yourself.

```
rem object number of character
character = 15
rem speed at which character will move
speed# = 3.0
mouseFlag = 0
moveFlag = 0

DO
  _mouse_click = mouseclick()

  rem if user right-clicks
  if _mouse_click = 2 and mouseFlag = 0
    mouseFlag = 1
    get3DMouse()
    moveFlag = 1
    rem get the angle between the character and targeted spot
    angle# = atanfull(targetX# - object position x(character), targetZ# - object position z(character))
    rem rotate character to point towards target
    yrotate object character, angle#
```

```
   endif

   if _mouse_click <> 2 then mouseFlag = 0

   if moveFlag = 1
      rem get character's new position
      x# = newxvalue(object position x(character), angle#, speed#)
      y# = getTerrainHeight()
      z# = newzvalue(object position z(character), angle#, speed#)
      position object character, x#, y#, z#
      rem if character is within range of target, then stop moving
      if getSquaredDistance#(x#, z#, targetX#, targetZ#) < (speed# * speed#)
         moveFlag = 0
      endif
   endif

   sync
LOOP
```

The **getSquaredDistance()** function returns the distance between two points. But instead of using **sqrt()** to get the actual distance, it returns the distance value still squared. We can get away with doing this because we're only using the distance for comparison. Just remember to square the other comparison value. If the character's speed is 3, then it'll stop moving if it's within 3 units of the target. As you can see, we only need to call **get3DMouse()** when the user clicks the mouse.

Step 3: Selecting units with the mouse

Ok, so now we know how to pick a targeted destination and move a character to that destination, but what good is it if we can't select the character? That's what this section will teach you to do. This is quite simple to do, and most of you can probably figure out how to do this in a matter of seconds and have it coded before I even finish writing this sentence. First, we have to draw the selection box on the screen. Before we can draw a box, we need to know two sets of coordinates first.

```
A_________
|        |
|        |
|________B
```

If we can determine A and B, then we know what are box is going to be. So lets find A. When a user clicks then drags the mouse, the box is formed from the spot initially clicked, to where the mouse cursor is currently positioned at. So point A is where the mouse was clicked first.

```
if _mouse_click = 1 and selectionFlag = 0
   selectionFlag = 1
   bx1 = mouseX()
   by1 = mouseY()
endif

if _mouse_click = 1 and selectionFlag = 1
   bx2 = mouseX()
   by2 = mouseY()
endif
```

You're probably thinking this is pretty simple, what's the catch? The catch is that this will only work if the user drags the selection in one specific direction, as from A to B as like the illustration above shows. If you dragged the mouse in the direction from B to A, you'll run into a problem. This next bit of code will check to see if a character is within the selection box. After you implement this, you'll see what the problem is.

```
rem object number of character
character = 15
```

```
cx = object screen x(character)
cy = object screen y(character)

if cx > bx1 and cx < bx2 and cy > by1 and cy < by2
   isSelected = 1
else
   isSelected = 0
endif
```

Do you see the problem yet? If not, check the IF statement above with these values:
bx1 = 200
by1 = 10
bx2 = 30
by2 = 500
cx = 100
cy = 300

Let's first look at the Y values. Is CY(300) greater than BY1(10) ? Yes, it is. Is CY(300) less than BY2(500)? Yes, so as far as we're concerned, the object is within the Y range of our selection box. Now let's check the X values. Is CX(100) greater than BX1(200)? No, but it's greater than BX2(30) and less than BX1(200). Even though the character is within the two BX values, it won't register as being so because the BX variables aren't in the proper order. So to fix this, we must order the coordinates which make up our selection box. What we want to is make sure that BX1 and BY1 are less than BX2 and BY2. Here is the full code for getting the two coordinates of the selection box, sorting the coordinates, and drawing a selection box.

```
_mouse_click = mouseclick()

if _mouse_click = 1 and selectionFlag = 0
   selectionFlag = 1
   bx = mouseX()
   by = mouseY()
endif

if _mouse_click = 1 and selectionFlag = 1
   bx2 = mouseX()
   by2 = mouseY()
endif

if _mouse_click <> 1 then selectionFlag = 0

if selectionFlag = 1
   rem sort the coordinates
   if bx < bx2
      bx0 = bx
      bx1 = bx2
   else
      bx0 = bx2
      bx1 = bx
   endif
   if by < by2
      by0 = by
      by1 = by2
   else
      by0 = by2
      by1 = by
   endif

   rem draw a box
   line bx0, by0, bx1, by0
   line bx0, by1, bx1, by1
   line bx0, by0, bx0, by1
   line bx1, by0, bx1, by1

   rem get the character's 2D screen coordinates
   cx = object screen x(character)
   cy = object screen y(character)

   rem if character is within the selection box..
   if cx > bx0 and cx < bx1 and cy > by0 and cy < by1
      isSelected = 1
   else
      isSelected = 0
   endif
endif
```

Step 4: Creating a real character

Now we need to determine what exactly our character can do. This is actually part of the planning stage of the game, and should be done before any code is even written. If you've never used a UDT (user-defined type) before, start learning. The design of this game relies on them and will greatly simplify your coding. So let's think about this, what are some things that every character in the game can do? move and attack. Now a worker character might be able to chop trees or harvest resources, but a soldier wouldn't, so for now we ignore those actions. Moving, attacking, and selecting the unit are the basic actions of every character, so lets create a UDT to define these.

```
type basicUnitActions
   isMoving     as boolean
   isAttacking  as boolean
   isSelected   as boolean
endtype
```

That takes care of basic actions, but what about attributes? What are the things that all characters have? They all have a name(soldier, worker, pilot), health, armor, damage, range, and speed. So create another UDT to define these properties. To clarify things, say you have 10 soldiers. These are the properties that all 10 soldiers will share, so don't include anything like position or current health. Those are properties that are dependent on each individual character.

```
type basicUnitProperties
   name          as string
   maxLife       as integer
   armor         as integer
   range         as integer
   speed         as float
endtype
```

Let's go ahead and create two character unit types now, a worker and a soldier. First, start off by creating a new array. The size of the array is the number of different character units that you'll have in your game. No need to use an array list here because you should know from the start (after your planning stage) how many different types there are.

```
dim characterType(2) as basicUnitProperties

#constant SOLDIER = 1
#constant WORKER = 2

characterType(SOLDIER).name = "Grunt"
characterType(SOLDIER).maxLife = 80
characterType(SOLDIER).armor = 4
characterType(SOLDIER).range = 0
characterType(SOLDIER).speed = 3.5

characterType(WORKER).name = "Peasant"
characterType(WORKER).maxLife = 45
characterType(WORKER).armor = 0
characterType(WORKER).range = 0
characterType(WORKER).speed = 2.25
```

The "Grunt" has a range of 0 because it'll fight with a sword. Hand to hand combat has no range, meaning the character has to be right next to the enemy in order to attack it. If the unit fights with a gun, then it would have a range value. Same thing applies to the "peasant", which also happens to have a 0 for armor as well, because it has none. Ok, now why did I define those two constants above? Because it'll make referring to the proper unit in the array much easier. Now we define an actual character. It would be nice if DBP could inherit UDTs, but it can't. So instead we must define a variable inside one type as another type and access its variables that way.

```
type character
   object        as integer
   action        as basicUnitActions
   isHarvesting  as integer
   targetX       as float
   targetZ       as float
   group         as integer
   unit          as integer
   life          as integer
endtype

dim characters(0) as character
```

Here, we initialize the array with 0 elements because it will be an array list, which can shrink and expand as we need to. Here's a list of the different properties mean.

object	- object number associated with this character
action	- basic unit action properties as defined earlier
isHarvesting	- intended use for only the peasant, 0 means not harvesting, other values refer different types of resources for which it is currently gathering
targetX	- if the unit is moving, these are the coordinates of where it is moving to
targetZ	- if the unit is moving, these are the coordinates of where it is moving to
group	- In other RTS games, you can select multiple units together and assign them a group and reselect them later by simply pressing that number on the keyboard. This holds the value for which group the unit belongs to
unit	- This is what type of character it is (solder, worker, etc...)
life	- How much life the unit currently has

Now lets see these UDTs in action.

```
rem create 5 soldiers
for t = 10 to 14
   load object "soldier.x", t
   array insert at bottom characters(0)
   index = array count(characters(0))
   characters(index).object = t
   characters(index).action.isMoving = 0
   characters(index).action.isAttacking = 0
   characters(index).action.isSelected = 0
   characters(index).isHarvesting = 0
   characters(index).group = -1
   characters(index).unit = SOLDIER
   characters(index).life = characterType(SOLDIER).maxLife
next t

rem create 2 workers
for t = 15 to 16
   load object "peasant.x", t
   array insert at bottom characters(0)
   index = array count(characters(0))
   characters(index).object = t
   characters(index).action.isMoving = 0
   characters(index).action.isAttacking = 0
   characters(index).action.isSelected = 0
   characters(index).isHarvesting = 0
   characters(index).group = -1
   characters(index).unit = WORKER
   characters(index).life = characterType(WORKER).maxLife
next t
```

Beginning to see how those **SOLDIER** and **WORKER** constants are making things easier? When initializing the variables for each character, we want each one to start out with its maximum available health. Instead of trying to remember which array element the soldier is, we just type **SOLDIER**. A reason for using UDT's and letting all characters of a certain type make reference to only 1 is because it makes updating the characters' properties much less intensive on the system. What does this mean exactly? Ok, say later in the game, the player has upgraded the armor of soldiers. If you had 1000 soldier characters and each one kept its own reference to its armor, you would have to loop through all 1000 characters and update its armor value. Instead, all you have to do here is update 1 single value like this: `characterType(SOLDIER).armor = 10`
Now all soldier type characters have an armor value of 10.

Step 5: Working with an array of characters in your game

After this step, you'll have something more like a real RTS game. So far, you've only learned how to select a single character, so let's change that. Above you learned how to create an array of characters, your army. The code below uses the same UDTs and arrays we defined earlier.

```
DO
  rem define a variable to hold mouse click status so only 1 system call is made each loop
  _mouse_click = mouseclick()

  rem if user right-clicks
  if _mouse_click = 2 and mouseFlag = 0
    mouseFlag = 1
    get3DMouse()
    rem acknowledge anything that needs to know if the user right-clicked for a new target destination
    assignTargetPositionFlag = 1
    rem get the angle between the character and targeted spot
    angle# = atanfull(targetX# - object position x(character), targetZ# - object position z(character))
  endif

  rem loop through all characters
  for char = 1 to array count(characters(0))
    rem get object number of this character for future object related actions
    obj = characters(char).object
    rem if user is currently drawing a selection box
    if selectionFlag = 1
      rem get the character's 2D screen coordinates
      cx = object screen x(obj)
      cy = object screen y(obj)
      rem if character is within the selection box, update isSelected variable
      if cx > selectionX1 and cx < selectionX2 and cy > selectionY1 and cy < selectionY2
        characters(char).action.isSelected = 1
      else
        characters(char).action.isSelected = 0
      endif
    endif
    rem if this character is selected
    if characters(char).action.isSelected = 1
      rem this is normally where you'd display the group number of the character
      rem but for this example, this will be used to show which characters are currently selected
      text object screen x(obj), object screen y(obj), "selected"
      rem  the user has right-clicked during this game loop..
      if assignTargetPositionFlag  = 1
        rem tell character to move
        characters(char).action.isMoving = 1
        rem assign the target coordinates to character
        characters(char).targetX = targetX#
        characters(char).targetZ = targetZ#
        rem rotate character to point towards target
        angle# = atanfull(targetX# - object position x(obj), targetZ# - object position z(obj))
        yrotate object obj, angle#
      endif
    endif
    rem if character is moving and hasn't reached its destination yet..
    if characters(char).action.isMoving = 1
```

```
      rem get current angle of character
      angle# = object angle y(obj)
      rem get speed of character
      speed# = characterType(characters(char).unit).speed
      rem determine new position of character
      x# = newxvalue(object position x(obj), angle#, speed#)
      y# = getTerrainHeight()
      z# = newzvalue(object position z(obj), angle#, speed#)
      rem position character at its new coordinates
      position object obj, x#, y#, z#
      rem get the target of this character
      tx# = characters(char).targetX
      tz# = characters(char).targetZ
      rem if character is within range of target, then stop moving
      if getSquaredDistance#(x#, z#, tx#, tz#) < (speed# * speed#)
        characters(char).action.isMoving = 0
      endif
    endif
  next char

  rem positions from new target have been assigned, switch flag off
  assignTargetPositionFlag = 0

  sync
LOOP
```

Obviously, this code is not complete to be considered a runnable program. It does, however, demonstrate how a single FOR loop through all our characters can be used to do all available character tasks. Selecting the unit, assigning a targeted destination, moving the unit, etc... From here, you should have a good idea on how to implement other tasks, such as harvesting resources or attacking.

Step 6: Creating groups

Most RTS games that I have seen have some form of grouping that allow you to quickly and easily select a group of characters, usually by pressing a number. This section is designed to show you how to do just that. First we must decide what the shortcut key will be for assigning a group number to a group of selected characters. For this, I will use CRTL + 1-9, which means the user presses the control key and a number between 1 and 9.

```
rem get scancode of key pressed
rem put this at the top of the code where the "_mouse_click = mouseclick()" is
_code = scancode()

rem reset flag
assignGroupFlag = 0
rem if user pressed control key
if controlkey() = 1
  select _code
    case 2 : _group_number = 1 : endcase
    case 3 : _group_number = 2 : endcase
    case 4 : _group_number = 3 : endcase
    case 5 : _group_number = 4 : endcase
    case 6 : _group_number = 5 : endcase
    case 7 : _group_number = 6 : endcase
    case 8 : _group_number = 7 : endcase
    case 9 : _group_number = 8 : endcase
    case 10 : _group_number = 9 : endcase
    case default : _group_number = -1 : endcase
  endselect
  rem switch on flag so the character loop knows to assign a group
  if _group_number > -1 then assignGroupFlag = 1
endif
```

Then in the code that loops through all your characters, inside the isSelected statement, assign the GN value for the characters' group number.

```
if characters(char).action.isSelected = 1
   if assignGroupFlag = 1
      characters(char).group = _group_number
   endif
```

Now that we can assign a group number to characters, I'll show you how to retrieve those groups,

```
rem reset flag
getGroupFlag = 0
rem make sure user isn't pressing control key and trying to actually assign groups rather than select
if controlkey() = 0
   select _code
      case 2 : _group_number = 1 : endcase
      case 3 : _group_number = 2 : endcase
      case 4 : _group_number = 3 : endcase
      case 5 : _group_number = 4 : endcase
      case 6 : _group_number = 5 : endcase
      case 7 : _group_number = 6 : endcase
      case 8 : _group_number = 7 : endcase
      case 9 : _group_number = 8 : endcase
      case 10 : _group_number = 9 : endcase
      case default : _group_number = -1 : endcase
   endselect
   rem switch on flag so character loops knows to select that group
   if _group_number > -1 then getGroupFlag = 1
endif
```

We only want to get the group if the user is not currently selecting any characters. This is part of the main character loop. Yet again, we are able to do multiple actions from within the same loop.

```
if selectionFlag = 1
   rem selection code here
else
   rem if user is trying to get a previously assigned group
   if getGroupFlag = 1
      rem check to see if this character is part of the group we want
      rem if not, deselect it incase it's currently selected
      if characters(char).group = _group_number
         characters(char).action.isSelected = 1
      else
         characters(char).action.isSelected = 0
      endif
   endif
endif
```

There is a small problem with this code. If you assign a group of characters the number 3, then try to assign a different group of selected characters the same number, both groups are now part of group number 3. They have been combined. If this is what you want, then leave the code as it is. However, if you want the previous group labeled as 3 to be released from that number when it is assigned to a new selection of characters, then this is how to do it:

```
if characters(char).action.isSelected = 1
   rem other code here
else
   rem if character is not selected and user is assigning a group number,
   rem then reset this character's unit value to -1 (no group)
   if assignGroupFlag = 1
      characters(char).group = -1
   endif
endif
```

Step 7: The grid

Here is a very important aspect of the game that players never actually see. It may not look like it, but RTS games are typically broken down into a grid. X squares by Y squares. Take your map or terrain or whatever, and break it up into a grid of squares. A house might take up 1 square, but a barracks might take up 4 squares. The size of each square should not be determined by the size of your map, but rather the size of your map should be determined by the number of squares in it. You determine the size you want each square to be. This is up to and how you want the scale of your game to be. Let's make each square 40x40 units. Now we decide how many squares, or tiles, we want to make up our map or grid. I'll make a small map of only 32x32 tiles. Create a new array to hold a spot for each map tile. Just multiply the numbers to get the real size of your terrain.

```
squareSize# = 40.0
mapX = 32
mapZ = 32
dim map(mapX,mapZ)
terrainX# = mapX * squareSize#
terrainZ# = mapZ * squareSize#
```

To make things visually easy for us, I'll create a matrix with the proper dimensions so that each tile in the matrix is 40units.

```
make matrix 1, terrainX#, terrainZ#, mapX, mapZ
```

That matrix will now have 32x32 tiles of each being 40units in size. So what's the point of making a grid like this? It's mostly to help with collision detection. A zero in the map array might mean that tile is empty. A 1 could mean that it is occupied by something, whether it's a building, character, or a tree. Another use of the grid is pathfinding, which you'll learn about later on. Now that you know what the grid is and how it's used, I'll be referring to the map array in future sections, such as this next topic...

Step 8: Let's chop it down!

Every RTS game has some way to gather resources used for creating and building its society. So in this example I will show how to chop down trees and gather lumber. This example will only show you how to target a tree for chopping and move your unit to it and begin gathering wood. It will not show you how to have your character automatically return to base with resources and walk back to the tree. Ok, first we must add a tree to our game. Just make a cylinder object and place it on the map. Make sure you know which tile it's in as part of the map array. Here's an easy way.

```
rem just doing the same thing here as we did with SOLDIER and WORKER
#constant TREE = 1
rem update map array so it knows a tree is at (8,3)
map(8,3) = TREE

rem make a tree
make object cylinder 5, 20
rem make sure we can see the object
set object cull 5, 0
rem position the tree inside the center of grid square (8,3)
position object 5, 8*squareSize#-(squareSize#/2), 10, 3*squareSize#-(squareSize#/2)
```

Here is the mouse interface design I will use for the resource targeting operation. When the user right-clicks on a resource(tree), that becomes the targeted resource. All selected units at that time will then move to that target and begin harvesting if that character's properties permits it to do so.

```
rem reset variable
_resource = 0
rem if user right-clicks
if _mouse_click = 2
   rem if user clicked on tree object
   _resource = pickResource()
   if _resource > 0
      rem resource has been targeted
   endif
endif
```

```
rem returns the index to the resource the mouse is on
function pickResource()
   for t = 1 to array count(resources(0))
      obj = resources(t).object
      check = pick object(mousex(), mousey(),obj,obj)
      if check > 0
         exitfunction t
      endif
   next t
endfunction 0
```

If you haven't guessed it already, we're going to need to create a new array of objects, and a new UDT as well. Just as we created the **characters()** array and **character** UDT, we need to do the same for resources.

```
type resource
   object             as integer
   quantityLeft      as integer
   resourceType  as integer
endtype

dim resources(0) as resource

rem create 1 tree
array insert at bottom resources(0)
index = array count(resources(0))
resources(index).object = 5
resources(index).quantityLeft = 100
resources(index).resourceType = TREE
```

If you want, you can create another UDT of basic resource properties, like we did for the character, to hold the default value for the resource quantity. The next step is assign the resource to all selected units. You'll also need to add a new variable to your **character** UDT at this time, call it **resoureQuantity** and define it as an integer.

Find this line of code in your character loop:

```
if characters(char).action.isSelected = 1
```

Then add this just below it:

```
if _resource > 0
   characters(char).isHarvesting = _resource
endif
```

But wait, can soldiers harvest lumber? We only want workers to harvest, so lets first check if the selected unit is a worker before assigning it a resource.

```
rem if player has selected a resource
if _resource > 0
   rem if this selected character is a worker
   if characters(char).unit = WORKER
      rem assign this character the resource
      characters(char).isHarvesting = _resource
```

We're not done yet, but we're getting there. Now we have to tell the character what to do when it is suppose to be harvesting.

```
rem if character has been assigned a resource
if characters(char).isHarvesting > 0
   rem get object number of resource
   resObject = resources(characters(char).isHarvesting).object
   rem if character is within range of resource (within 15 units)
   if getSquaredDistance#(object position x(obj), object position z(obj),object position x(res), object position z(res)) < 225
```

```
  rem make sure character hasn't already gathered from this resource within the last 1 second
    if characters(char).lastAction+1000 < timer()
      rem increase character's gathered resource quantity
      characters(char).resourceQuantity = characters(char).resourceQuantity + 1
      rem reset character's time of last action it did (harvested)
      characters(char).lastAction = timer()
      rem decrease quantity of resource
      resources(resID).quantityLeft = resources(resID).quantityLeft - 1
      rem if resource object has no more resources
      if resources(resID).quantityLeft <= 0
        characters(char).isHarvesting = 0
      endif
    endif
  endif
endif
```

Yes, we do have to add yet another variable to the **character** UDT, **lastAction** an integer. Beginning to see how easy it is to add new properties to our characters? Now what this variable does is it keeps track of how long ago the character has completed its last action. If we incremented the characters resource quantity by 1 for as long as it was in range of the resource, it could end up gathering hundreds within 1 second! We don't want that, so in the code above we make sure the character can't gather any resources any faster than 1 per second. This variable doesn't have to be limited to simply regulating resource gathering. Think of an archer that fires arrows at targets. Do you want it to fire 500 arrows every second? Unless it's using an automatic crossbow with rapid fire, I don't think so. Also, one last thing to remember is you must decrement the quantity from the resource itself, otherwise it would never run out. And, finally, we check to see if the resource has been depleted. If so, remove the resource ID from the character.
There are two different ways you can gather resources. In games like Warcraft and Age of Empires, a worker goes to the resource, collects a specific amount, then returns to base to drop of the resources where it is then added to stockpile. This is how I've set it up. Another way of doing it is to just add the collected resource immediately to your stockpile. The worker never returns to base to drop it off. One thing I haven't shown here in this section is how to handle resource carrying limits. Suppose a worker can only carry 20 lumber at a time. When its **resourceQuantity** reaches 20, it must return to base before it can continue. I'll leave that one up to you to figure out. It's not a difficult task, and if you've learned anything from this article, then you should be able to see how to implement new actions for characters. Attacking units can be set up the same way as gathering resources if you think about it.

Step 9: Working with animation

Up until now, I've only used cubes and spheres to represent characters. Using animated models might sound like something you would do near the end of your game's creation to polish up the look of it, but there is a little code involved in making those models animated the proper frames at the right time. This section is intended to teach you how to sort that all out in an easy way. The model I will be using from now on is the dwarf model created by Psionic.
http://www.psionic3d.co.uk

To start things off, you need to add a new integer variable to the **character** UDT called **stage**. This will store the animation state of the object and tells us if it's walking, standing idle, or animating some other action. Now to create a new UDT called **basicUnitAnimations**.

```
type basicUnitAnimations
  idleStart      as integer
  idleEnd        as integer
  walkStart      as integer
  walkEnd        as integer
  action1Start   as integer
  action1End     as integer
  action2Start   as integer
  action2End     as integer
endtype
```

We can add more stages to this later as we need them. Now add an **animations** variable to the **basicUnitProperties** UDT of that new animation UDT.

```
type basicUnitProperties
   name          as string
   maxLife       as integer
   armor         as integer
   range         as integer
   speed         as float
   animations    as basicUnitAnimations
endtype
```

Now all you have to do is set the frame numbers for each. If you downloaded the dwarf model in a zip file originally from Psionic, then you have the list of frames for each animation in a text file. If not, here are the proper animation frames. I added them to the WORKER character.

```
characterType(WORKER).animations.idleStart = 326
characterType(WORKER).animations.idleEnd = 359
characterType(WORKER).animations.walkStart = 1
characterType(WORKER).animations.walkEnd = 13
characterType(WORKER).animations.action1Start = 111
characterType(WORKER).animations.action1End = 125
```

And for easier use later on in the code, define a few constants so you can remember animation stages better.

```
#constant ANIMATE_IDLE = 1
#constant ANIMATE_WALK = 2
#constant ANIMATE_ACTION1 = 3
#constant ANIMATE_ACTION2 = 4
```

Now on to the real code. I wrote a function to easily set an object's animation loop given the character's index in the array and stage of animation to switch to as inputs.

```
rem set object's animation loop
function setAnimation(char as integer, stage as integer)

   rem if object's animation is already set for this stage, then exit
   if characters(char).stage = stage then exitfunction

   unit = characters(char).unit

   if stage = ANIMATE_IDLE
      loop object characters(char).object,characterType(unit).animations.idleStart, characterType
                  (unit).animations.idleEnd
      characters(char).stage = stage
   endif
    if stage = ANIMATE_WALK
      loop object characters(char).object,characterType(unit).animations.walkStart, characterType
                  (unit).animations.walkEnd
      characters(char).stage = stage
   endif
    if stage = ANIMATE_ACTION1
      loop object characters(char).object,characterType(unit).animations.action1Start, characterType
                  (unit).animations.action1End
      characters(char).stage = stage
   endif
    if stage = ANIMATE_ACTION2
      loop object characters(char).object,characterType(unit).animations.action2Start, characterType
                  (unit).animations.action2End
      characters(char).stage = stage
   endif
endfunction
```

This function is called once for each character, so just stick it inside the FOR loop at the end. The next and final task is merely a trivial matter; where and when should the character's stage change? At the beginning of the FOR loop, set a variable to hold the character's default stage of animation.

```
for char = 1 to array count(characters(0))
   rem reset variable to default animation stage
   stage = ANIMATE_IDLE
```

Let's start with when the character should change to its walking animation. If the character **isMoving**, then it should be looping through the walk animation. `stage = ANIMATE_WALK` Understand it so far? Now, for chopping lumber I used the animation sequence of **ANIMATE_ACTION1**. So set **stage** to equal this where your code checks to see if the character is currently harvesting lumber. Remember, the character only harvests resources if it's within range. Now, when you get to the end of the character loop and call the setAnimation(char, stage) function, it will switch the character's animation loop sequence, if it has to, to match the current state of action of the character. But what happens when the resource runs out and the character stops harvesting? Well, since the character automatically stops gathering its resources(or lumber from the tree), the `stage = ANIMATE_ACTION1` line of code is never reached, and therefore, **stage** never changes from its original assignment of the idle animation. So then when the **setAnimation()** function is called, it'll see that the current state of animation should be idle instead of the character's previous state of action. It's really quite simple if you think about it.

Step 10: Formations

By formations, I'm referring to the order of how the units are grouped together when walking. Save you have 10 soldiers walking towards a target. How should they look as they walk? All in one single line, or perhaps you want them to form a solid box shape. Or maybe you want them to form a circle as they move. Whatever the pattern you want, this step of the tutorial will show you how to program different formations. How we actually select them in the game won't be shown until I cover the in-game interface.

The first formation I will show you is the block formation. The key to creating successful formations is to allow the formation to take shape regardless of how many characters are selected. You could hard-code each position into an array and apply it that way, but I don't like that because, it's inflexible. Look at the image below. The numbers indicate the positions of each character which will make up the block formation.

1	2	5	10
4	3	6	11
9	8	7	12
16	15	14	13

The numbers go in a spiral because when the number of characters is unknown this will keep the most block-like formation. Now how do we assign the positions? When the action to assign the formation to all selected units is called, all we have to do is keep track of the count of how many units we have. So in the character main loop where we check to see if a character is selected, we increment a counter and assign its position. If it's the first character we've found selected, it's number 1. The next loop may find another, but the counter is at 2. So looking at our grid we know where to place this unit as well. Now to put it all into code. I thought the best way to do this was to create a function that took in a number and saved the grid location into 2 global variables (column,row).

```
Global G_formationX
Global G_formationY
Global G_middleX#
Global G_middleY#

function blockFormation(n as integer)
  dec n, 1

  ring = sqrt(n)
  idx = n - (ring*ring)

  if idx >= ring
    formationY = ring
    formationX = 2*ring - idx
  else
    formationX = ring
    formationY = idx
  endif
endfunction
```

Note that this function returns rows and columns starting at (0,0).
I would make another function called `getFormation()` and create a few Constants to make things easier later on when we start getting multiple formations to choose from.

```
#constant FORMATION_BLOCK = 1

function getFormation(unit as integer, formation as integer)
  if formation = FORMATION_BLOCK then blockFormation(unit)
endfunction
```

This function will determine the mid-point between all selected characters. It is needed to determine where the formation should be located.

```
function getMeetingCoordinates()
  lowX#=999999
  lowZ#=999999
  highX#=0
  highZ#=0
  rem loop through all characters
  for char = 1 to array count(characters(0))
    if characters(char).action.isSelected = 1
      obj = characters(char).object
      if object position x(obj) < lowX# then lowX# = object position x(obj)
      if object position z(obj) < lowZ# then lowZ# = object position z(obj)
      if object position x(obj) > highX# then highX# = object position x(obj)
      if object position z(obj) < highZ# then highZ# = object position z(obj)
    endif
  next char

  rem assign coordinates to the global variables
  G_middleX# = lowX# + (highX#-lowX#)/2.0
  G_middleZ# = lowZ# + (highZ#-lowZ#)/2.0
endfunction
```

This function must be called once when you have chosen to assign a new formation.
An alternative to the method above, you can have the formation form at the position of the first selected character. That would prevent the need of having to do the extra character loop.

Now, to add the code from above into the main character loop. For the formation counter I mentioned before, if you're code has been set up like mine throughout this tutorial, then you can just use the `_selected_unit` count variable. We'll still need a new variable that states whether a formation has been selected or not. Until I cover how to do an interface, you can just use the space bar to indicate a call to the formation.

```
if spacekey()=1 then _formation = FORMATION_BLOCK : getMeetingCoordinates()
```

Because the units must move to create the formation, they'll abandon their previous orders such as harvesting or an old target location. Remember to reset the necessary variables.

```
    if characters(char).action.isSelected = 1
       rem keep count of how many units are currently selected
       inc _selected_count, 1
       if _formation > 0
          rem assign formation to character
          characters(char).formation = _formation
          rem call the construction of the formation
          getFormation(_selected_count, _formation)
          rem throw flag so that character gets assigned its new target coordinates
          assignTargetPositionFlag = 1
          rem assign new target coordinates for this character
          targetX# = G_formationX*squareSize# + G_middleX#
          targetZ# = G_formationZ*squareSize# + G_middleZ#
          rem reset previous orders
          characters(char).isHarvesting = 0
          characters(char).action.isAttacking = 0
       endif
```

Notice how the coordinates are made to match the formation. `squareSize#` allows the characters to form their block formation but only occupy 1 grid square of the map. This is only effective if the grid squares of your physical map are larger than your actual characters. Once the characters are in formation, we'll need to remember that, so its time for a new variable inside our character UDT.

```
formation as integer
```

Now to make sure those units keep formation when you move them. When you right-click and assign a new destination for your characters, check to see if its part of a formation. If so, offset the target so that it keeps its formation with the group. Modify the previous code where you assign the target position for the formation.

Change:

```
          rem assign new target coordinates for this character
          targetX# = G_formationX*squareSize# + G_middleX#
          targetZ# = G_formationZ*squareSize# + G_middleZ#
```

To:

```
          rem assign new target coordinates for this character
          targetX# = G_middleX#
          targetZ# = G_middleZ#
```

Then in the block where you assign the target position, add this:

```
        assignTargetPositionFlag = 1
        rem tell character to move
        characters(char).action.isMoving = 1
        rem check to see if character belongs to a formation
        offsetX# = 0
        offsetZ# = 0
        if characters(char).formation > 0
           getFormation(_selected_count, characters(char).formation)
           offsetX# = G_formationX*squareSize#
           offsetZ# = G_formationZ*squareSize#
        endif
        rem assign the target coordinates to character
        characters(char).targetX = targetX#+offsetX#
        characters(char).targetZ = targetZ#+offsetZ#
        rem rotate character to point towards target
        angle# = atanfull((characters(char).targetX - object position x(obj)), (characters(char).targetZ - object position z
(obj)))
```

Some things to keep in mind, this code is setup to work with formations set a certain way. Each formation you create must be broken down into a grid. When getFormation() is called, the proper global variables are assumed to have been initialized that give the correct row and column of the formation for the character to follow. As long as those guidelines are followed, you can create any kind of weird formation you want.

Step 11: Path-finding, at last!

Once you implement this into your game, it'll start to feel more like a real, playable game, rather than just mindlessly moving around characters. The concepts in this step may be hard for some of you to follow, but I'll do my best to illustrate what's going on in the code. I will not be showing you how to code your own A* path-finding algorithm, there is sufficient data on the internet for you to learn that yourself. This step is intended to show you how to implement a pre-existing library. The A* library that I'll be using was written by IanM of the Apollo forums, thanks IanM. To date, I believe he has written the most efficient and easy to use path-finding library, which is why I have chosen to use it. The dba file format of the library should be included within the zip file which contained this tutorial document. It is well documented, and I suggest you read over his comments before reading on here to implement it into our RTS.

First thing to do is to include the library into your program. Put the line of code below at the beginning of your program.

```
#include “a star library.dba”
```

One more thing I should mention, before I continue, is that this library will not consider terrain to be passable by some and not passable for others, such as water. (boats versus soldiers) It can be done with alteration to the library file, but that is beyond the scope of this tutorial.

Now we create our search map. There is only 1 map which needs to be created. All our characters might have different paths, but all are part of the same map.

```
CreateSearchMap(mapX, mapZ)
```

Remember from the beginning we set up the variables **mapX** and **mapZ** to both equal 32 and used to create the matrix/terrain. For simplicity reasons which you will see shortly later, our search map will be the same size. The command will create a search map with 32 by 32 tiles, each tile initially setup to passable. This next command creates the number of different paths that we store. This number is really up to you and the design of your game. Think of how many characters that you could be moving at once. This could use up a lot of memory, but it will take up little CPU time. If the terrain is static, meaning it doesn't change, then it would be a logical decision. However, our terrain will be dynamic, allowing the search map to change in real-time. Because of this, the characters will be searching for a path the whole time it is moving towards it target. We actually only need 1 path and I will explain why. As we loop through each moving character, we calculate a path. We assign a temporary target of the first square in the path. Until the character reaches that new tile, or square, it will not have to search for a new path. We do the same thing for each character in the loop, each time updating the single path. The characters do not need to remember there whole path because we only move the characters 1 tile at a time, and each time it reaches a new tile, it recalculates the path to find the next tile it can move to. If that didn't make sense, then you should have a more clear understanding of how it works once I show you the code.

```
CreateSearchPathLists(0)
```
(0 does count as 1 element)

Add these lines into your **character** UDT.

```
   tempX          as float
```

```
  tempZ         as float
```

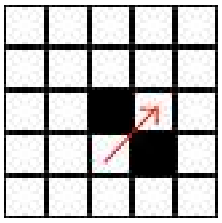
Those variables will hold the temporary target position of each character. Initialize those variables with -1. This will mean that no temporary target is assigned and that we need to search the map for a new path. We only check for new paths and update the temp variables if the character is moving. So within the appropriate part of the character loop, first translate the target position into a grid position.

```
targetTileX = int(characters(char).targetX / squareSize#)
targetTileZ = int(characters(char).targetZ / squareSize#)
```

Do the same for the character's current position. Now, that we have the beginning and end of the path we want, it's time to search for a suitable path. Which search to choose to use is completely up to you.

```
SearchMapAStar8(0,currentX, currentZ, targetTileX, targetTileZ)
```

This function, **SetSearchRestrictDiagonals(Mode)**, means that the path will not move diagonally if the two side tiles are blocked.



If set to the default value of 0, the path can walk diagonally through obstacles as shown in the picture. Otherwise, the above move would be illegal and the path would form around the blocks. As with the search type, this is entirely up to you and how you want your game to work.

Assign the new temporary target position. The **GetSearchPath()** functions return the map's grid coordinates, so convert them into world coordinates.

```
characters(char).tempX = GetSearchPathX(0,1)*squareSize#
characters(char).tempZ = GetSearchPathY(0,1)*squareSize#
```

The next stage is to modify the movement code that we've already written. Only perform the above code when no temporary target is set.

```
if characters(char).tempX = -1
```

No need to check tempZ because both variables would be set to -1 simultaneously.

```
angle# = atanfull((characters(char).tempX - object position x(obj)), (characters(char).tempZ - object position z(obj)))
```

Remove the equation to find the angle between the character and target from the previous **IF** block it was located in (when you assigned the targeted position) to the new **IF** block you just created above. Since your character will no longer be walking in a straight line to reach its target, the angle will have to change and be recalculated each time a new temporary target tile is set. When the character has reached its final destination and stops moving, remember to reset the **tempX** and **tempZ** variables back to **-1**. If the character is not within range, then check to see if it is within range of the temporary target. If so, then set those variables to -1.
Before, you checked to see if the distance from the character to the final destination was within range. Change it from the final target to the temporary target. If the character is within range, check to see if the current tile the character is in is also the final targeted tile. If so, we can stop moving, else we set the temp variables to -1 and it updates the path and new temporary target.

```
if getSquaredDistance#(x#,z#,characters(char).tempX,characters(char).tempZ) < (speed# * speed#)
  targetTileX = int(characters(char).targetX/squareSize#)
  targetTileZ = int(characters(char).targetZ/squareSize#)
  currentX = int(object position x(obj)/squareSize#)
  currentZ = int(object position z(obj)/squareSize#)
  if (targetTileX = currentX) AND (targetTileZ = currentZ)
    characters(char).action.isMoving = 0
  else
    characters(char).tempX = -1
    characters(char).tempZ = -1
  endif
endif
```

There's still one last thing we must change in our pre-existing code, the harvesting section. When the player selects a resource for a character to harvest, instead of setting that direct location as the target, find a clear tile surrounding the resource and mark that as target instead. This will prevent the characters from standing directly on top of the resource. Also, note that because of the way we implemented our characters to form into specific formations, nothing extra is needed to make them use path-finding when forming the pattern. One final thing to mention is that so far, the map is completely clear and passable at all tiles. To change that, use the following command to set a tile as impassible.

```
SetSearchMap(X,Y,Value)
```

Go ahead and run your code now to make sure everything is working as it should be. To test the path-finding, I setup a matrix in wireframe form to show the imaginary grid of the search map. You should now be able to build a complex map and watch your characters navigate through it.

Step 12: Unknown

Something…..