TBGL user defined entities

Introduction to the technique

Petr Schreiber

TBGL programming

for beginners

1st June 2010

Introduction

ThinBasic module for accelerated graphics, TBGL, offers multiple ways to represent the scene. The most straightforward one is often referenced as entity system.

Entity system allows organization of various kinds of entities in scenes. Programmers can define "solid" objects using geometric primitives, m15 models or display lists and specify their relationship. This is enough for most typical applications, but what to do when the programmer want to use different kinds of objects.

The answer is – use the function slot entity.

What is function slot

Function slot is the approach provided by the module allowing programmers to define their own, custom entities. All the geometry based entities in TBGL share some common properties and actions.

- Position, rotation, scale
- Colour, texture information
- User defined data
- Hierarchy
- Name

When creating user defined entity, it is safe to presume having all these options will be enabled for it. How to define the look? How to use the 3rd party rendering functions? How to actually create the custom entity?

Custom entity, based on function slot approach, can be created in the following steps:

- define user defined type describing entity additional properties
- create entity of type "function slot"
- set it user data of choice
- write the function which will render custom primitive
- write action functions to control specific properties of the entity

Example

To demonstrate this simple approach, the technique will be explained on very basic example. Let's presume we need special kind of entity – simple heat fan. It can be set to 2 states - turning, or stopped.

The first thing we need to do is design our data structure:

```
TYPE tFanEntity
  workingState  AS BYTE
END TYPE
```

The first thing which is evident is that we don't need to specify position or other general things – this is common for all geometric entities, so we just define the special properties, in this case working state.

Then we create the function slot entity, presuming our custom function will be represented by function *Fan_Render*. After the creation we supply some initial custom data to it.

```
DIM FanData AS tFanEntity
FanData.workingState = 0

TBGL_EntityCreateFuncSlot( %sScene, %eFan, 0, "Fan_Render")
TBGL_EntitySetUserData(%sScene, %eFan, FanData)
```

The function determining fan look will need to render some objects to represent the object in 3D. It should also handle the animation of the fan. When working state is set to 1, it will turn, in other case the propeller will be rendered still, stopped.

```
SUB Fan_Render()

  ' -- Retrieve data
  DIM Data AS tFanData AT TBGL_EntityGetUserDataPointer(%sScene, %eFan)

  ' -- If in rotation state, let's turn
  IF Data.WorkingState = 1 THEN
    TBGL_EntityTurn(%sScene, %eFan, 0, 0, 90/FrameRate)
  END IF

  ' -- Fan represented by two crossed thin boxes
  TBGL_Box(0.1, 2.0, 0.5)
  TBGL_Box(2.0, 0.1, 0.5)

END SUB
```

Now we have guaranteed once calling *TBGL_SceneRender*, the *Fan_Render* routine will be used to draw our custom fan entity.

As was mentioned earlier, we need to supply some more functions to be able to control the fan.

Doing this is very intuitive, all we need to do is get access to our fan data, and modify them as we need. All the changes to these properties can be then later reflected in the rendering routine.

The fan is simple object, so we will just need to control whether it turns or not. For this, we declare simple user *function* or *sub* with parameters, which will internally affect the fan properties.

```
SUB Fan_SetWorkingState( flag AS BYTE )

  ' -- First we overlay structure over the memory of fan entity
  DIM Data AS tFanData AT TBGL_EntityGetUserDataPointer(%sScene, %eFan)

  ' -- Then we can set the properties, which will be immediately updated to entity
  Data.WorkingState = flag

END SUB
```

With this routine, just by calling *Fan_SetWorkingState(1)* fan will be turned on, and with *Fan_SetWorkingState(0)* it will immediately stop rotating.

Thinking more generally

It seems we have everything we need, but there is one problem with the solution proposed above – all actions are hardwired to *%sScene* scene and *%eFan* entity. This might be enough when we have only one entity of the kind and using one scene, but as we know, this is not the most common case.

How to make the functions usable for all entities of type "fan"? We need to properly identify the calling entity then we will be able to retrieve its custom data.

This is done by *TBGL_CallingEntity* command in conjunction with *TBGL_tEntityIdentifier* structure. It might sound complicated, but have a look at modified rendering routine below with changes marked in green:

```
SUB Fan_Render()

  DIM element AS TBGL_tEntityIdentifier AT TBGL_CallingEntity
  ' -- Retrieve data
  DIM Data AS tFanData AT TBGL_EntityGetUserDataPointer(element.scene, element.entity)

  ' -- If in rotation state, let's turn
  IF Data.WorkingState = 1 THEN
    TBGL_EntityTurn(element.scene, element.entity, 0, 0, 90/FrameRate)
  END IF

  ' -- Fan represented by two crossed thin boxes
  TBGL_Box (0.1, 2.0, 0.5)
  TBGL_Box (2.0, 0.1, 0.5)

END SUB
```

When using this approach, variable *element* will always be filled with contextual *scene ID* and *entity ID*. This allows using *Fan_Render* procedure for unlimited number of entities of fan type, each with completely independent custom data.

Modifying *action* functions is even simpler, just by adding new parameters.

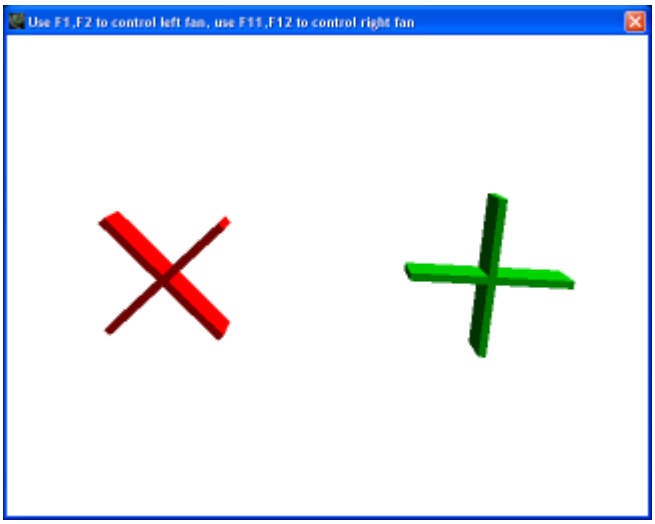
```
' -- Our custom actions
SUB Fan_SetWorkingState( scene AS LONG, entity AS LONG, flag AS BYTE )

  ' -- First we overlay structure over the memory of fan entity
  DIM Data AS tFanData AT TBGL_EntityGetUserDataPointer(scene, entity)

  ' -- Then we can set the properties, which will be immediately updated to entity
  Data.WorkingState = flag

END SUB
```

And this is it, we have completed the creation of custom entity.

Just a side note – unlike the case of the rendering function, which is fired internally by the *TBGL_SceneRender* command, we cannot use *TBGL_CallingEntity* here. This is because the entity control procedures, such as *Fan_SetWorkingState* in this case, are freely called directly by the programmer.

Conclusion

We have demonstrated the basics of creating custom entities, how to create custom rendering routine and how to modify the entity properties.



Pic. 1 Screenshot from the sample script showing two fan entities

The full source code can be found and studied by opening the file from ThinBasic installation: *SampleScripts/TBGL/EntityBased/TBGL_CustomFanEntity.tBasic*

The demonstrated example is very simple, so it is packed with other code in single file. For serious projects we recommend to organize custom entities in separated include files, which you can later easily incorporate to your project, and move from one application to another easily.