



GAME MECHANICS KIT PROGRAMMER'S GUIDE

The first part of this guide will help Torque 3d engine programmers to install Game Mechanics Kit (GMK) into an existing Torque based project.

The second part covers the basics of working with game objects' templates for use with GMK Editor.

[Visit Game Mechanics Kit Forum](#) if this guide doesn't answer all of your questions.

GMK installation instructions

To be able to use GMK features you must merge your current Torque Engine code with the one of GMK.

Before performing any merging make sure you have a backup copy of all of your source files!

Merging process consists of the following steps:

1. Choosing appropriate GMK distribution.
2. Merging C++ source files.
3. Adding GMK's C++ files to your project file (*.vcproj).
4. Updating scripts and data files.

1. Choosing appropriate GMK distribution

Each GMK distribution designed to be used with specific Torque version only.

Make sure that you're updating your version with appropriate distribution of GMK. For example if you are using *TGEA 1.7.1* you should patch it only with *GMK for TGEA 1.7.x*.

Note

There are precompiled versions of *exe* files for each supported engine version. So if you don't want to recompile *exe* for yourselves you can move straight to N4 “Updating scripts and data files”. *Exe* files are already compiled with **Physics Pack** support (*Bullet* for *T3D* and *TGEA* and *ODE* for *TGE*). The only thing you have to do to activate physics is Step 7 of **Physics Pack** installation.

2. Merging C++ (*.h, *.cpp) source files

If you want to start a new GMK based project, you will just need to copy files from GMK package to your Torque Engine folder and overwrite all files with the same name.

But if you have modified original engine code by yourself you should merge your code with GMK. To simplify merging process you can use a CodePatch utility.

You can download CodePatch from [this page](#).

CodePatch is using three-way merge (also know as merging from a common ancestor) algorithm to merge text files. CodePatch compares files from three folders each one contains a different modification of the engine. The first one is your current engine version (should be specified by “**my path**”). The second folder is the unmodified source code from the original Torque installation (“**base path**”, probably will looks like this “*c:\TGEA_1_8_1\engine*”). If you have overwritten sources with your own modifications you’ll need to install Torque engine to new folder and provide CodePatch with appropriate path to original engine sources. Third is the folder with sources of GMK (“**their path**”). CodePatch will compare files from “my” path and “their” relatively to “base” and place merged file in the result folder (specified by “**result path**”).

Some times CodePatch will not be able to merge your code automatically. This situation is known as a *conflict*. It happens when changes in files from “my” and “their” folders contradict to each other. By default CodePatch will simply put both changes in result file for you to handle it manually later. In this case you should look for '<<<<<<<' string (conflict descriptions starts with that) in result sources.

All GMK changes in code are commented with special marks, so you can easily identify such blocks for example:

```
//logicking >>
addField("gameMechanicsMode", TypeBool, Offset(mGameMechanicsMode, WorldEditor));
addField("showIcons", TypeBool, Offset(mShowIcons, WorldEditor));
//logicking <<
```

The other way to handle conflicts is to set “**Use external program to resolve conflicts**” checkbox and provide CodePatch with path to the WinMerge in the “**Program path**” field.

3. Adding GMKs C++ files to your project file

To finish the integration of GMK C++ code you should make changes to your Visual Studio project file. Actually you have to add GMK *.cpp and *.h files to your project.

Warning!

At this stage do **not** add files from either *engine\source\T3D\logickingMechanics\physics\bullet*, *engine\source\T3D\logickingMechanics\physics\ode* and *engine\source\T3D\logickingMechanics\physics\physX* directories.

Here's list of files to add*:

```

engine\source\T3D\logickingMechanics\aiBot.h
engine\source\T3D\logickingMechanics\aiBot.cpp
engine\source\T3D\logickingMechanics\visualEffect.h
engine\source\T3D\logickingMechanics\visualEffect.cpp
engine\source\T3D\logickingMechanics\physics\physics.cpp
engine\source\T3D\logickingMechanics\physics\physics.h
engine\source\T3D\logickingMechanics\physics\physicsBody.cpp
engine\source\T3D\logickingMechanics\physics\physicsBody.h
engine\source\T3D\logickingMechanics\physics\physicsDummy.cpp
engine\source\T3D\logickingMechanics\physics\physicsDummy.h
engine\source\T3D\logickingMechanics\physics\physJoint.h
engine\source\T3D\logickingMechanics\physics\physShape.cpp
engine\source\T3D\logickingMechanics\physics\physShape.h
engine\source\T3D\logickingMechanics\physics\physShapeDummy.cpp
engine\source\T3D\logickingMechanics\physics\physShapeDummy.h
engine\source\T3D\logickingMechanics\physics\ragDoll.cpp
engine\source\T3D\logickingMechanics\physics\ragDoll.h
engine\source\T3D\logickingMechanics\physics\rigidBody.cpp
engine\source\T3D\logickingMechanics\physics\rigidBody.h
engine\source\T3D\logickingMechanics\physics\softBody.cpp
engine\source\T3D\logickingMechanics\physics\softBody.h
engine\source\T3D\logickingMechanics\logickingMechanics.h (only for TGEA 1.7.1 and TGE 1.5.2)
engine\source\T3D\logickingMechanics\refBase.h (only for TGEA 1.7.1 and TGE 1.5.2)
engine\source\T3D\logickingMechanics\refBase.h (only for TGEA 1.7.1 and TGE 1.5.2)

```

* For TGE you should read "*engine\game\logickingMechanics*" instead of "*engine\source\T3D\logickingMechanics*".

For TGE 1.5.2, TGEA 1.7.1 and TGEA 1.8.1 add

```
engine\source\console\arrayObject.cpp **
```

** It's Daniel Neilsen's script array class resource -

<http://www.garagegames.com/community/resources/view/4711>

4. Updating scripts and data files

Updating scripts are nearly identical to merging source code. If you're starting a brand new project you only have to overwrite files from "game" folder of your project with the ones of GMK. Otherwise CodePatch will be in use again. This time you should provide paths to the "game" folder

in your project, the original base project, and game folder within GMK to “my path”, “base path” and “their path” respectively.

The installation process is over now. If every thing has gone smoothly you should be able to compile your engine from source codes and play with GMK features and editor.

Installation of Physics Pack

By default GMK uses a *dummy physics* library. It means you can place rigid bodies and ragdolls in the editor and they will do nothing but stay still.

To get GMK physics features working you need to install **Bullet** (*TGEA, T3D*), **ODE** (*TGEA, TGE*) or **PhysX** (*T3D*) physics library.

If you are *TGEA* user and don't sure what to choose use **Bullet**. For *TGE* owners the only option is **ODE** since we were unable to get **Bullet** allocators work with *TGE* memory manager.

If you are *T3D* user you can choose to use **PhysX**. In this case you have to compile T3D with **PhysX** support and then move directly to **Step 5** of **Physics Pack** installation.

Warning! In case if you want to use **PhysX** with your project.

Please make sure that you are able to compile and run your T3D project with **PhysX** support **before** any GMK installation. You have to install **PhysX SDK** as well. Refer to T3D documentation and forums at GarageGames.com for more info.

Next steps will show you how to setup physic library on your project. GMK Physics Pack includes precompiled versions of *.lib* files to link with your engine. But if you feel confident enough you can get [Bullet](#) or [ODE](#) library sources from their official websites and compile them by yourselves.

Step 1

Bullet

Copy *physicsPack\bullet* directory to *engine\lib* directory.

ODE

Copy *physicsPack\ode* directory to *engine\lib* directory. (*TGEA*)

Copy *physicsPack\ode* directory *lib* directory. (*TGE*)

Now you will need to change properties of your project in Visual Studio. You can access properties by pressing Alt+F7.

Step 2

Provide path to additional library directories.

Project Properties->Linker->General->Additional Library Directories

Bullet

../../../../engine/lib/bullet/lib

ODE

../../../../engine/lib/ode/lib (TGEA)

../lib/ode/lib (TGE)

Step 3

Provide names of libraries files to link.

Project Properties->Linker->Input->Additional Dependencies

Bullet

debug: *libbulletcollision_d.lib libbulletdynamics_d.lib libbulletmath_d.lib libbulletsoftbody_d.lib*

release: *libbulletcollision.lib libbulletdynamics.lib libbulletmath.lib libbulletsoftbody.lib*

ODE

debug: *ode_singled.lib*

release: *ode_single.lib*

Step 4

Provide path to include files.

Project Properties->C/C++->General->Additional Include Directories

Bullet

../../../../engine/lib/bullet/include

ODE

../../../../engine/lib/ode/include (TGEA)

../lib/ode/include (TGE)

Step 5

Add files C++ to your project.

Bullet

Add files from *engine\source\T3D\logickingMechanics\physics\bullet*

engine\source\T3D\logickingMechanics\physics\bullet\physicsBullet.cpp
engine\source\T3D\logickingMechanics\physics\bullet\physicsBullet.h
engine\source\T3D\logickingMechanics\physics\bullet\physJointBullet.cpp
engine\source\T3D\logickingMechanics\physics\bullet\physJointBullet.h
engine\source\T3D\logickingMechanics\physics\bullet\physShapeBullet.cpp
engine\source\T3D\logickingMechanics\physics\bullet\physShapeBullet.h

ODE

Add files from engine\source\T3D\logickingMechanics\physics\ode*

engine\source\T3D\logickingMechanics\physics\ode\physicsODE.cpp
engine\source\T3D\logickingMechanics\physics\ode\physicsODE.h
engine\source\T3D\logickingMechanics\physics\ode\physJointODE.cpp
engine\source\T3D\logickingMechanics\physics\ode\physJointODE.h
engine\source\T3D\logickingMechanics\physics\ode\physShapeODE.cpp
engine\source\T3D\logickingMechanics\physics\ode\physShapeODE.h

* For TGE you should read “*engine\game\logickingMechanics\physics\ode*”

PhysX

Add files from engine\source\T3D\logickingMechanics\physics\physx

engine\source\T3D\logickingMechanics\physics\physx\physicsPhysX.h
engine\source\T3D\logickingMechanics\physics\physx\physicsPhysX.cpp
engine\source\T3D\logickingMechanics\physics\physx\physJointPhysX.cpp
engine\source\T3D\logickingMechanics\physics\physx\physJointPhysX.h
engine\source\T3D\logickingMechanics\physics\physx\physShapePhysX.cpp
engine\source\T3D\logickingMechanics\physics\physx\physShapePhysX.h
engine\source\T3D\logickingMechanics\physics\physx\physShapeSoftPhysX.h
engine\source\T3D\logickingMechanics\physics\physx\physShapeSoftPhysX.cpp
engine\source\T3D\physics\physx\pxInteriorInstance.cpp
engine\source\T3D\physics\physx\pxInteriorInstance.h

Step 6

Open file engine\source\T3D\logickingMechanics\physics\physics.h.

Bullet

Uncomment line with

```
#define PHYSICS_BULLET
```

ODE

Uncomment line with

```
#define PHYSICS_ODE
```

PhysX

Uncomment line with

```
#define PHYSICS_PHYSX
```

Step 7

Open script file scriptsAndAssets\server\scripts\logickingMechanics\physics.cs

Comment with // line

```
$GMK::Physics::Lib = "Dummy";
```

Bullet

Uncomment line

```
$GMK::Physics::Lib = "Bullet";
```

ODE

Uncomment line

```
$GMK::Physics::Lib = "ODE";
```

PhysX

Uncomment line

```
$GMK::Physics::Lib = "PhysX";
```

Step 8

Rebuild your project in your VC++.

Game Objects and Templates

Term *game object* stands behind interactive dynamic entities of the Torque simulation like triggers, AI bots, players, doors, weapons, inventory items etc. Basically games objects are the objects that game designers are using to create gameplay mechanics. In contrast *non-game objects* usually don't change themselves during the game and are used by level designers to create static scenes. Typical examples of such objects are interiors, terrain, water objects etc.

GMK is designed to work exclusively with *game objects*.

Template is a description of game object for GMK Editor. Template provides editor with information on how specific game object should be created and edited.

Let's start with an example of *EventTrigger* object template.

File: server\scripts\logickingMechanics\eventTrigger.cs

```
//-----
// for Game Mechanics Editor
//-----
activatePackage(TemplateFunctions); // opening package with template functions
registerTemplate("EventTrigger", "Triggers", "EventTrigger::create(EventTrigger);"); //register template
setTemplateField("EventTrigger", "ignoreAI", "1", "bool", "Misc", " Check true to ignore AI characters.");
// events
setTemplateEvent("EventTrigger", "onEnter", "", " %this - trigger object. %arg0 - is an object that enters the trigger. ");
setTemplateEvent("EventTrigger", "onLeave", "", " %this - trigger object. %arg0 - is an object that leaves the trigger. ");
setTemplateEvent("EventTrigger", "onTick", "", " %this - trigger object. ");
deactivatePackage(TemplateFunctions); // close package with template functions
// action
setTemplateAction("EventTrigger", "setEnabled", "(%isEnabled)");
```

Template Basics

All template function are stored in the *TemplateFunctions* package, so every operation on templates must start with `activatePackage(TemplateFunctions)` and end with `deactivatePackage(TemplateFunctions)`.

To create a new template you should call *registerTemplate* function:

```
registerTemplate("EventTrigger", "Triggers", "EventTrigger::create(EventTrigger);");
```

First argument is a name of template, second is a category name, and third code for creation of template object instance.

Default categories are *Abstracts*, *AI*, *Triggers*, *Breakables*, *Objects*, *Doors*, *PhysicalZones*, and *Misc*.

You can add new categories by yourself like this

```
addCategory("AI", "tools/missionEditor/images/LogickingEditor/AI");
```

First argument is a name of category, second – icon name prefix. There are two types of icons for each category a big (64x64) and a small (24x24) with “_ico” suffix. Both icons must be in PNG format.

Categories description is in

```
scriptsAndAssets\server\scripts\logickingMechanics\gameObjectCategories.cs
```

“*Abstracts*” is special category that used for creation of abstract templates that can’t have instances. Concrete templates can inherit from abstract (or any other) templates, to share abilities. You can look how it’s done with *DamageTrigger*.

```
scriptsAndAssets\server\scripts\logickingMechanics\damageTrigger.cs
```

DamageTrigger inherits most of its template code from EventTrigger template.

```
inheritTemplate("DamageTrigger", "EventTrigger");
```

inheritTemplate should be always call **before** *registerTemplate*.

Template Fields

To add properties to template call *setTemplateField* function.

```
setTemplateField("EventTrigger", "ignoreAI", "1", "bool", "Misc", " Check true to ignore AI characters.");
```

First argument is a name of template, second is a name of field (dynamic or static) of the object’s class, third is field type (can be *bool*, *string*, *list*, *file* or *vector*), fourth – default value of the field, fifth - the name of group field belongs to and the last – short description of field.

Events and Actions

Event-reaction is simple but powerful mechanism of game objects’ interaction. Game objects evoke events on certain conditions. For example *EventTrigger* raises “onEnter” and “onLeave” events when player enters or leaves trigger area respectively. Evocation of event can be done in C++ or TorqueScript by calling *signal* function. For trigger’s “onEnter” it looks like this:

File: server\scripts\logickingMechanics\eventTrigger.cs

```
function EventTrigger::onEnterTrigger(%this, %trigger, %obj)
{
    if(!%trigger.isEnabled()) return;
    if(!%obj.playerControlled && %this.ignoreAI) return;
    %trigger.signal("onEnter", %obj); //evoking "onEnter" event with object which enters trigger as an argument
    Parent::onEnterTrigger(%this, %trigger, %obj);
}
```

In your game you can have a multitude of EventTrigger objects. Each object can has its unique reaction on “onEnter” event. Basically *reaction* is arbitrary string of Torque script code and it stored as dynamic field right in the object creation code in the mission file.

This part of template description makes “onEnter” event visible to GMK editor:

```
setTemplateEvent("EventTrigger", "onEnter", "echo(\"put code here\")", "%this - trigger object. %arg0 - is an object that enters the trigger. ", 1);
```

First argument is name of template, second - name of event.

Third argument is default code and fourth is short description of arguments passed to event from when it was evoked with signal function (the arguments of event are always named with %this, %arg0, %arg1, etc). The last argument is a number or arguments passed to the event excluding %this.

To make life for game designers even easier template description includes descriptions of *actions*. Action is a member function of the game object.

```
setTemplateAction("EventTrigger", "setEnabled", "(%isEnabled)");
```

First parameter is name of template, second - name of the action, third – description of formal arguments list for the action call.