GAME MECHANICS KIT DESIGNER'S GUIDE

This document covers the basics of working with the Game Mechanics Kit editor and templates for game designers.

Visit Game Mechanics Kit Forum if this guide doesn't answer all of your questions.

What is the Game Mechanics Kit?

Game Mechanics Kit (*GMK*) is an add-on for GarageGames' *TGE*, *TGEA* and *T3D* engines. *GMK* provides functionality and tools for game designers to script and edit the mechanics of the game development process in a very easy and visual way. *GMK* implements principle of building blocks: create your game with ready made components.

GMK consists of two main parts: *game object templates* and *editor*.

Game Object Templates

Core of *GMK* is the set of C++ and TorqueScript template classes (or simply *templates*) that enhance functionality of original Torque engine. Template objects inherit general design patterns that present in one form or another almost in every 3D game. These templates include but not limited by the following categories of objects:

* Simple AI bot, with easy configurable and controlled behavior.
* Interactive environment: destructible and explosive items, doors, switches, lockers etc.
* Different variations of a trigger object.
* Visual effect objects.
* Inventory items.

* Auxiliary objects: counters, invisible walls etc.

The good point to start learning templates is to look at **Orcs Rule** game from within GMK Editor.

Game Mechanics Editor

Game Mechanics Kit Editor (GMK Editor) is created for game designers in the first place. GMK Editor provides functionality to create game objects and change their parameters in a very user friendly way. Designers will be able to create complex gameplay situations with a minimal knowledge of Torque Script. GMK Editor adds some useful functions to original World Editor, but its major advantage is implementation of "event-reaction" system for game objects. So the user can configure objects' parameters and interaction with other objects right within the editor.

How to start?

You can access GMK Editor by selecting *Game Mechanics Kit* field from the original Torque editor menu and then selecting *Game Mechanics Kit Editor Toggle*. You can also toggle between original World Editor and GMK Editor by pressing the *Ctrl+E* combination.

Editor Toolbar

 Majority of the *GMK* editor functions can be reached through *Toolbar* window located at the bottom of the editor's window.
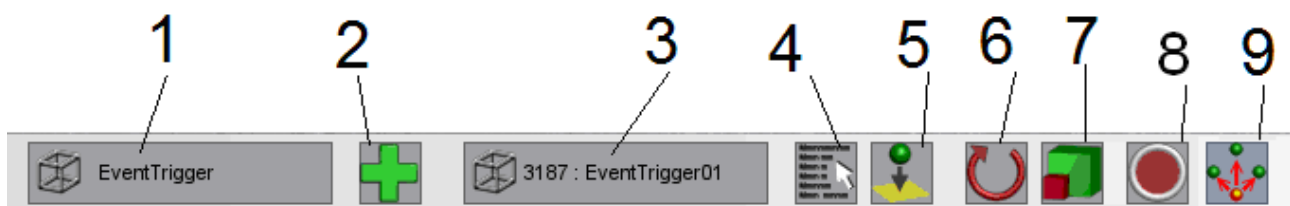

Figure 1 GMK Editor Toolbar.

1. Opens *Object Templates List.*
2. *Add button* creates new objects with chosen template.
3. Opens *Object Properties* window for selected object.
4. Opens *Game Object List* - a list of all available game objects in simulation.
5. "Drops" selected object to the ground.
6. "Rotation Mode" - you can rotate object by mouse, and you **don't** have to hold ALT key at the same time.
7. "Scale Mode" - you can scale object by mouse, and you **don't** have to hold CTRL+ALT keys at the same time.
8. Shows or hides *GMK* icons for game objects.

9. Shows or hides events' links of game objects.

How to use editor?

Imagine that we want to create a simple combination of game objects: AI bot attacks player whenever player walks into the trigger.

First we need to create trigger and AiBot objects. Using *Object Templates List* window we must select corresponding templates and create new *SpaceOrcBot* and *EventTrigger* objects. Newly created objects can be seen in *Game Object List*.
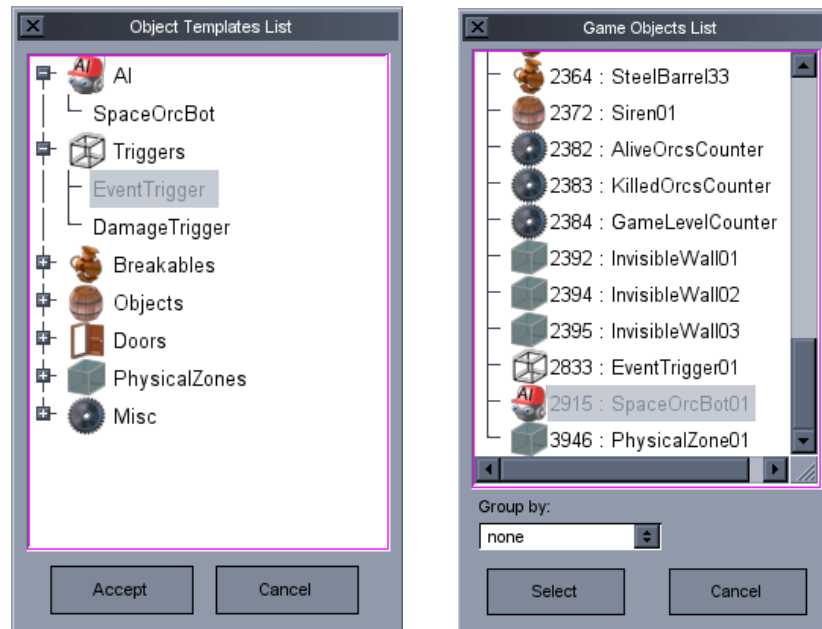

Figure 2 Object Template List and Game Objects List windows.

Then in *Object Properties* window we need to modify behavior of the trigger. *EventTrigger* object has an "onEnter" event that evokes every time player character collides with trigger's physical shell. The designer needs to provide a reaction for this event.
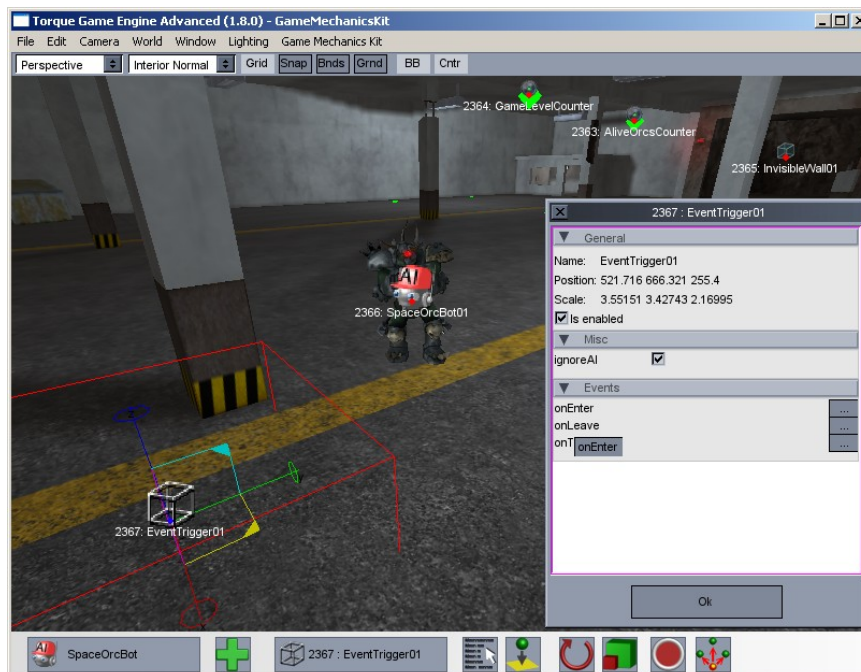

Figure 3 GMK Editor window with Object Properties window opened for EventTrigger01.

Basically reaction is arbitrary piece of TorqueScript code. So you can write following: "SpaceOrcBot01.setEnemy(%arg0);", SpaceOrcBot01 is a name of AiBot object, and %arg0 is an object that represent entity, that collides with the trigger object.
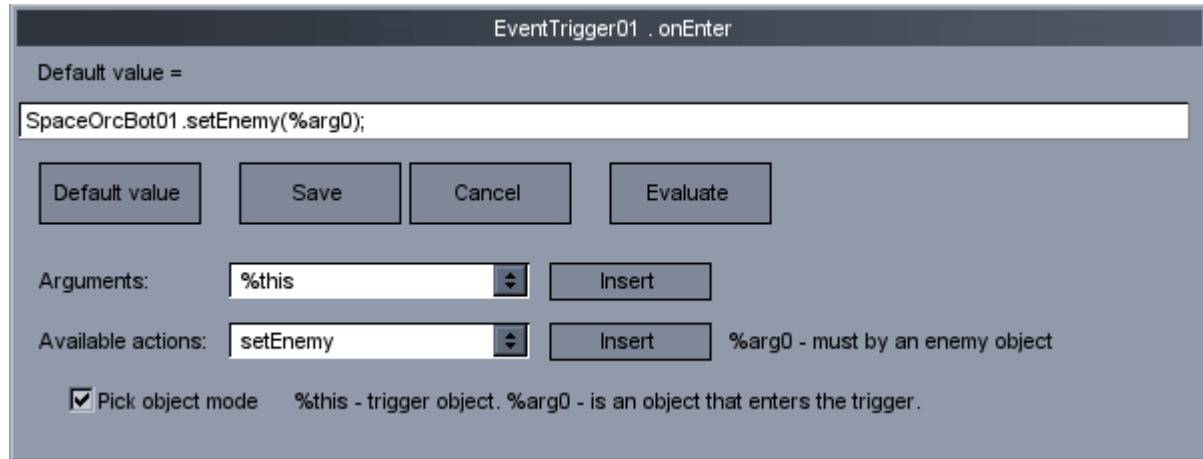

Figure 4 Event Edit window for "onEnter" event.

But the GMK Editor can help write this code for you. In the *Event Edit* window with "pick mode" flag set you can enter object's name by just clicking on the appropriate object in the editor window. And when the object is selected you can choose an available action from the list.

That's all! Now you need to save you mission exit editor and start to play. The bot will attack you, immediately after you enter the trigger.

Description of GMK Object's Templates

AiBot

***Description*:**
>   *AiBot* can act as an opponent or ally of player character.

***Examples of usage*:**
- Enemies attack player trying to prevent him to accomplish certain goal.

***Datablock fields*:**
>   *maxForwardSpeed, maxBackwardSpeed, maxSideSpeed* – speed of movement for different directions in meters per second.

>   *deathSnd* –sound that will be played on bot's death.

>   *painSndCount, painSnd[0], painSnd[1]…* - count and sounds for pain. Pain sounds plays when bot receives damage.
>   *ragdoll* – datablock for ragdoll. If specified Bot will turn in ragdoll when dies.

>   *weapon* – datablock for weapon image, that bot own.
>   *ammo* – type of used ammo.
>   *ammoCount* – amount of ammo.
>   *shootingDelay* – *delay* between shot in milliseconds.

>   *chaseFarDist* - if enemy is further than *chaseFarDist*, bot will begin chase.
>   *chaseCloseDist* - if enemy is closer than *chaseCloseDist*, stop chasing.
>   *attackFov* – Field of view in degrees. Bot will attack enemy when enemy is within this FOV.
>   *attackDist* – maximum dist that allows bot to attack enemy.

>   *strafeMinDist, strafeMaxDist* – bot will strafe left and right on the random distance within these values.
>   *strafeChangeDirTime* – time in milliseconds when bot will change direction of strafe.

***Template fields*:**
>   *health* – bot's health.
>   *item* – a dataBlock name for an item that bot spawns on his death.

>   *perceptionEnabled* – If set to *true*, bot will automatically attack every enemy – anyone form *opposite team* – within the *viewDist* radius.

>   *viewDist* – Bot's maximum view dist. Bot will see no enemies outside this radius.

>   *ourTeam* – Bot's team name. To set bot as an ally to player should be "players", as a foe to player, should be "bots".

>   *oppositeTeam* - Rival to bot's team. Bot will attack anyone, who has its *ourTeam* equal to bot's oppositeTeam. To make bot attack players characters should be set to *"players"*, to make bot attack other bot - *"bots"*.

***Commands*:**

setEnemy *(enemy)* – assigns new enemy to bot.

followPath(*path*, *movementType*) – Set the path for the bot to follow. *Path* – is an Path with waypoints.
MovementType can be one of following:
$Path::TO_TARGET - Moving to target node and stopping there,
$Path::CYCLE - cycling though nodes in path,
$Path::PATROL - going to the first and finish nodes, changing direction on both ends,
$Path::RANDOM - wandering to random nodes.

setGuard(*objectToGuard*) - assigns object, that bot will guard. Object can be player's character or other bot.

***Events*:**

onSpawn – when but just created.

onDamage – when bot receives damage. *%arg0* - who cast damage.

onAlert – when new enemy is assigned to bot. *%arg0* - enemy.

onDeath – bot dies.

onPathNodeReached – bot reaches node when following the path. *%arg0* - node index in the path.

onPathEnd – bot reaches last node of the path.

onNoEnemy – no more enemies to attack.

Breakable

***Description*:**

Breakable is an object that can break and/or explode: barrel full of powder, fuel can or fragile wooden chair. When someone or something hits the breakable it health decreases. When health is lower then specified bound breakable starts to burn (if applicable) and after some time explodes. Destruction of *breakable* is a combination of animation played by breakable's shape and attached particle effect with sound.

***Datablock fields*:**

timeToBreak – time for *breakable* to burn before explosion.
timeToBreakVariance – random variance of *timeToBreak*.
timeToVanish – time before debris of breakable vanish.
breakAnimation – name of animation for break or explosion.
preparatoryEffect – effect of burning.
breakEffect – effect of explosion or breaking.

Template fields:

> *health* – Value of health decreases when breakable receive damage.

> *healthBound* – bottom level of health, when breakable starts to burn.

Commands:

> prepareExplosion*()* – explodes object. If object has preparatory effect it starts burning first;

Events:

> *onValueChanged* – whenever internal value is changed

> *onLowerBound* – when lower bound is reached.

> *onUpperBound* – when upper bound is reached.

> *onBound* – when upper or lower bound is reached.

Counter

Description:

> *Counter* is an object that increments or decrements internal integer value accordingly to incoming commands. Counter evokes events whenever its value changes or reaches specified bound.

Examples of usage:

- Load next location when all players of the group mode have entered some trigger. Can be useful for some cooperative multiplayer gameplay.
- Open a certain door when number of bots killed is 10.

Template fields:

> *value* - internal integer value, which is affected by commands.

> *minValue* – lower bound.

> *maxValue* – upper bound.

Commands:

> *inc()* – increase internal value by one;
> *dec()* – decrease internal value by one;
> *set(val)* – sets internal value to *val*;

Events:

> *onValueChanged* – whenever internal value is changed

> *onLowerBound* – when lower bound is reached.

> *onUpperBound* – when upper bound is reached.

> *onBound* – when upper or lower bound is reached.

CutScene

***Description*:**
>   *Cut scenes* are important part of many modern computer games. A *cut scene* is a sequence over which the player has no control over the situation. Cut scenes used to advance the plot, present character development, and provide background information, atmosphere, dialogue and clues.

***Examples of usage*:**
- Introduce player with a new monster in a dramatic fashion.
- Show some action (the door is opening), which happens in response to a player use of a certain lever.

***Template fields*:**
>   *chunks* – list of script chunks that will execute during *cut scene* play.
>   *sounds* – list of sounds that will play with cut scene.
>   *paths* – list of camera paths.

***Commands*:**
>   play*()* – plays *cut scene*.
>   stop*()* – stops *cut scene*.
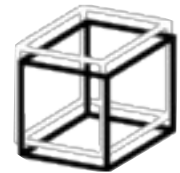
***Events*:**
>   *onPlay* – when scene starts playing.

>   *onStop* – when scene finishes playing or user has interrupted *cut scene*.

DamageTrigger

***Description*:**
>   *DamageTrigger* is used to prevent characters from entering specific areas by damaging or killing them.

***Examples of usage*:**
- Damage player whenever he steps on lava.
- Kill player when he fall to the bottom of some canyon.

***Datablock fields*:**
>   *tickPeriodMS* – Controls how often trigger will apply damage for collided characters. The default value is 100 MS.

***Template fields*:**
>   *ignoreAI* – Check *true* to ignore AI and reacts only on player controlled characters.

>   *damage* – Amount of damage player will receive every *tickPeriodMS*.

>   *damageType* – Type of the damage. Use it when you want to handle different source of damage like fire, fall etc.

***Commands*:**
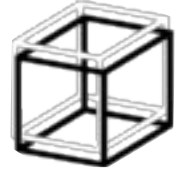>   *setEnabled(isTrue)* – enables or disables trigger;

***Events*:**
>   *onEnter* – when character enters the trigger.

>   *onLeave* – when character leaves the trigger.

onTick – evokes in a specified interval of time.

EventTrigger

Description:
> *EventTrigger* is a very common object for game designers. It evokes an event every time character (player or NPC) enters the trigger area.

Examples of usage:
- Activate enemies, when player reaches certain area.
- Start a cut scene, when player enters the room.

Datablock fields:
> *tickPeriodMS* – Controls how often trigger will check collision with characters. The default value is 100 MS.

Template fields:
> *ignoreAI* – Check *true* to ignore AI and reacts only on player controlled characters.

Commands:
> *setEnabled(isTrue)* – enables or disables trigger;

Events:
> *onEnter* – when character enters the trigger.
>
> *onLeave* – when character leaves the trigger.
>
> *onTick* – evokes in a specified interval of time.

InvisibleWall

Description:
> *InvisibleWall* is used to prevent player or AI reaching certain areas.

Examples of usage:
- Place *InvisibleWall* to block player from ability to fall to the canyon.
- *InvisibleWall* on the seashore stops player from entering the water.

Template fields:
> *ignoreAI* – check *true* to ignore AI and reacts only on player controlled characters.

Commands:
> *setEnabled(isTrue)* – enables or disables *InvisibleWall*.

Openable (Doors, Lockers and Switches)

*Description***:**

    Openable is base template for all objects that can be opened or closed.

*Template fields***:**

    isLocked – *openable* object can't be opened while this value is *true*.

    keyToUnlock – a data block of key object. When player uses *openable* and has right key in its inventory *openable* will unlock and open.

    isOpen – initial state of *openable*. Opened when *true*, closed otherwise.

*Commands***:**

    open() – tries to open, unless not locked;
    close() – closes openable;

*Events***:**

    onOpening – when begins to open.
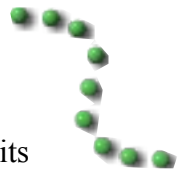
    onOpened – when successfully opened.

    onClosing – when begins to close.

    onClosed – when successfully closed.

Path

*Description***:**

    Path is order list of waypoints. *Waypoint* – is a point in 3D space presented by its position and rotation.

*Examples of usage***:**

- Path for camera fly in cut scenes.
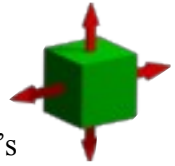- Path for *AiBot* movement.

*Template fields***:**

    isLooping – path will be treated as cyclic with movement from last to first waypoints.

    speed – speed of camera movement.

    markers – list of waypoints object that make a path.

Physics Objects (Rigid Bodies, Soft Bodies)*

*Description***:**
>   *Physics Objects* are objects affected by physics forces like gravity or explosion's impulse.

*Examples of usage***:**
- Stock of crates that blocks player's path.
- A dead body simulated by ragdoll physics object.

*Datablock fields***:**
>   *shapeFile* – filename of the *dts* model.
>   *scale* – vector for the shape.
>   *mass* – a mass of the object in kilograms.

>   Fields that allow user to add physics collision shape to the object:

>   *shapeType*** – determines type of the physics collision shape, can be following: *$ShapeType::Box*, *$ShapeType::Sphere*, *$ShapeType::Capsule*, *$ShapeType::Cylinder*.
>   *rotAngles* – Euler angles in degrees, for rotation of collision shape relative to model.
>   *offset* – offset vector for collision shape relative to model.

>   Fields for softbodies:

>   *poseMatchKoef* – Softness of body (from 0 to 1). 1 is to have a rigid body, 0 - for cloth.
>   *attachedPointsNum*, *attachedPoints [0], attachedPoints [1]…* - positions in shape local coordinates where soft body will be "nailed" to.

>   * Soft bodies available only for **bullet** physics library.
>   ** Shape size will be calculated automatically from the original model bounding box. When using **bullet** you can see physics geometry in debug mode. Just call from console **physicsDraw(1)**.

>   Fields for sounds of physics interactions:

>   *minContactSpeed* - minimum speed of physical interaction when physics object produce collision sound. The higher you set this value, the more powerful interaction will produce sound while less powerful will remain silent.

>   *slidingThreshold* - physics object can either slide (roll) or collide. Engine will calculate dot product of collision normal with object velocity. If absolute value of this dot is closer to 1 then assumed that object is colliding, closer to 0 – sliding. You can setup different sound for the colliding and sliding.

>   *collisionSoundsCount*, *collisionSound [0], collisionSound [1]…* - count and sounds for collision. Chooses random sound when object collides.

>   *slideSoundsCount*, slideSound *[0]*, slideSound *[1]…* - count and sounds for sliding. Chooses random sound when object slides or rolls.

***Template fields*:**

***Commands*:**
> *setEnabled (isTrue)* – enables or disables physics simulation for current object;

SoundPlayer

***Description*:**
> *SoundPlayer* is used to play random sound or music without specifying sound profile in scripts beforehand.

***Examples of usage*:**
- Play action music (2D sound), when player starts to fight with enemies.
- Play cyclic sound (3D) of waterfall. *SoundPlayer* object itself must be placed appropriately.

***Template fields*:**
> *filename* – name of the sounds file (.ogg or .wav). Not used when *sfxProfile* field used.
> *volume* – volume of the sound.
> *loop* – when *false* sound will be played only once.
> *is3d* – when sound is 3D the position of *SoundPlayer* matters and sound is mono, otherwise sound is 2D and can be stereo.
> *refDist, maxDist* – Controls volume of the sound in relevance to listener (user) position. Only when sound is 3D. When listener closer to *SoundPlayer* then *refDist* – the volume will be at max. When distance if greater the *refDist* volume will decrease till vanish completely at *maxDist*.
>
> *preload* – when *true* sound will be load to memory immediately after *SoundPlayer* creation.
> *description* – use a predefined audio *SFXDescription* like *AudioDefault3d* or *AudioClose3d*. In this case you won't need to use *is3d, refDist, maxDist, volume* and *loop* fields.
> *sfxProfile* – use a fully defined by *SFXProfile* sound from scripts. In this case you won't need any of the fields including filename.

***Commands*:**
> play*()* – plays sound.
> stop*()* – stops sound.
> *pause()* – pauses sound. Can be resumed with *play()*.